UCRL -ID 108725

Techniques, Processes, and Measures for Software Safety and Reliability

D. Sparkman

May 30, 1992
Version 3.0

Lawrence Livermore National Laboratory

Disclaimer

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Techniques, Processes, and Measures for Software Safety and Reliability

D. Sparkman

Manuscript date: May 30, 1992
Version 3.0

Contents

Techniques, Processes, and Measures for Software Safety and Reliability

1. Technical Summary

Presented herein is a summary and comparison of domestic and international standards pertaining to software reliability and safety. The standards are reviewed for similarities and differences and are supplemented with views collected from experts in the field of high-integrity software. There is a growing consensus among experts that the development of high-integrity software requires the application of good software engineering processes and techniques, supplemented by measurements of their effectiveness and of the reliability of the product.

Several standards focus on safety planning and analysis. These are IEEE P1228, MIL-STD 882B, and MoD 00-55. Others focus more on general quality assurance planning and activities. These include DoD 2167A and 2168, FAA 13B, 16A and 18A, IEC 65A(Secretariat)122, IEC 880, and MoD 00-55. Some recommend specific safety-related analyses. Failure modes and effects analysis are recommended by IEEE 352 and 577 and IEC 65A(Secretariat)122), system hazard analysis by MIL-STD 882B, MoD 00-56, IEC 65A(Secretariat)122, and software-specific hazard analysis by MIL-STD 882B, MoD 00-56.

Several standards define classification schemes for hazard severity or software safety integrity levels and make recommendations on practices to apply on the basis of a system's classification level. The standards that employ such schemes are MIL-STD 882B, IEC 65A(Secretariat)122, and MoD 00-55 and 00-56. The new revision of IEEE 1012 may also incorporate such an approach.

A unique set of standards is the pair consisting of IEEE 982.1 and 982.2, which contain of a set of definitions of 39 measures, many of which are applicable to reliability measurement, and guidance for their use. AFSCP 800-14 also identifies a subset of quality indicators.

The majority of standards recommend using a lifecycle-phased approach to software development wherein certain quality increasing and safety evaluation practices are applied within each phase . The planning activities recommend include using software quality assurance plans, configuration management plans, verification and validation plans, and coding standards. Also recommended are technical reviews and audits, the use of well-defined error reporting procedures, performing risk management (from the point of view of project failure), and conducting safety analyses. Only three standards make specific recommendations for the use of formal methods (mathematical specification and proof techniques); these are IEC 880, IEC 65A(Secretariat)122 and MoD 00-55. IEC 65A(Secretariat)122 and IEC 880 also make recommendations either for the selection of specific programming languages or for the acceptable properties of programming languages.

IEC 65A(Secretariat)122, which is not yet finalized and is in preliminary use, and MIL-STD 882B are the most specific standards and may be the most highly regarded with respect to the development of software for use in safety-related applications. IEEE P1228 also specifically addresses this class of software. Since this standard will not be finalized until late 1992, its adequacy is unproven.

2. Purpose

The purpose of this report is to provide a detailed survey of current recommended practices and measurement techniques for the development of reliable and safe software-based systems. This report is intended to assist the United States Nuclear Reaction Regulation (NRR) in determining the importance and maturity of the available techniques and in assessing the relevance of individual standards for application to instrumentation and control systems in nuclear power generating stations. Lawrence Livermore National Laboratory (LLNL) provides technical support for the Instrumentation and Control System Branch (ICSB) of NRR in advanced instrumentation and control systems, distributed digital systems, software reliability, and the application of verification and validation for the development of software. This report responds to FIN L-1867, Project II.

3. Scope

This report includes an evaluation of domestic and international, industry and government accepted practices and standards relating to software reliability and safety-related software systems. This information is supplemented with viewpoints from experts in these fields. Many processes, practices, and techniques for estimating and predicting software reliability are discussed in these standards and publications. This report specifies the processes, practices, and techniques recommended by the reviewed standards. It also provides a description of these processes, practices and techniques.

This interim report incorporates AFSCP 800-14, DoD 2167A and 2168; FAA 013B, 016A, and 018A; IEC 65A(Secretariat)122, 880, and 1014; IEEE 279, 352, 577, 730, 982.1, 982.2 and P1228; MoD 00-55 and 00-56; MIL-STD 882B, and the referenced publications. The international and domestic standards arena contains additional standards related to safety-related systems that are not included in this report. These standards include DOE 5480.5, 5480.6 and 5481.1B; EIA SEB6 and SEB6A; HMSO Parts 1 and 2; IEC 65A(Secretariat)96; IEE 5; IEEE 603 and 627; and MIL-STD 1574A.

4. Report Organization

This report contains four major sections. The first, Section 6, discusses the issue of reliability in a safety-related software product. The second, Section 7, discusses software reliability and quality practices that span multiple life cycles. Section 8 discusses safety issues derived from the standards reviewed. Section 9 details many techniques for estimating the reliability of a software product. This section is subdivided into the life cycle phases, each phase discussing the software reliability techniques associated with that life cycle phase. Section 9 also identifies several software reliability measures or metrics that can be used with each technique. Appendix A contains a matrix of the major techniques and processes identified in Sections 7 and 10; Appendix B the safety processes and procedures from Section 8; Appendix C the specific reliability and safety techniques identified in Section 9; and Appendix D contains brief descriptions of the indicator and predictor measures for the techniques in Section 9.

5. Definitions and Acronyms

5.1 Definitions

Important definitions based on current and draft standards and published literature have been included to assist in the comprehension of this report. When more than one applicable definition is available, all have been included.

accident. (1) An unplanned event or series of events that results in death, injury, illness, environmental damage, or damage to or loss of equipment or property. *IEEE P1228 Draft E, July 1991.* (2) An unintended event or sequence of events that causes death, injury, environmental or material damage. *MoD 00-56 Defence Standard, May 1989.*

availability. (1) The degree to which a system or component is operational and accessible when required for use. Often expressed as a probability. *IEEE 610-1976.* (2) The excepted fraction of time during which a software component or system is functioning acceptably (Musa, Iannino, and Okumoto, 1988).

concept phase. (1) The period of time in the software development cycle during which user needs are described and evaluated through documentation (for example, statement of needs, advance planning report, project initiation memo, feasibility studies, system definition, documentation, regulations, procedures, or policies relevant to the project). (2) The initial phase of a software development project, in which the user needs are described and evaluated through documentation (for example, statement of needs, advance planning report, project initiation memo, feasibility studies, system definition, documentation, regulations, procedures, or polices relevant to the project). *IEEE 610.12-1990.*

data encapsulation. See encapsulation.

design phase. The period of time in the software life cycle during which the designs for the architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements. *IEEE 610.12-1990.*

encapsulation. A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module. *IEEE 610.12-1990.*

event. A significant happening that may originate in the environment or the system. *MoD 00-56 Defence Standard, May 1989.*

failure. (1) The termination of the ability of a functional unit to perform its required function. (2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is encountered. *IEEE 610.12-1990.*

fault. (1) An accidental condition that causes a functional unit to fail to perform its required function. (2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonymous with bug. *IEEE 610.12-1990.*

firmware. The combination of a hardware device and computer instructions and data that reside as read-only software on that device. Notes: (1) This term is sometimes used to refer only to the hardware device or only to the computer instructions or data, but these meanings are deprecated. (2) The confusion surrounding this term has led some to suggest that it be avoided altogether. *IEEE 610.12-1990.*

formal method. Mathematically based method for the specification, design, and production of software. Also includes a logical inference system for formal proofs of correctness and a methodological framework for software development in a formally verifiable way. *MoD Defence Standard 00-55, formal mathematical method.*

formal proof of correctness. A way of proving, by a mathematical proof using formal rules, that a computer program follows its specification. *MoD Defence Standard 00-55.*

implementation phase. The period of time in the software life cycle during which a software product is created from design documentation and debugged. *IEEE 610.12-1990.*

installation and check-out phase. The period of time in the software life- cycle during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required. *IEEE 610.12-1990.*

measure. A quantitative assessment of the degree to which a software product or process possesses a given attribute. *IEEE 610.12-1990.*

operation and maintenance phase. The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements. *IEEE 610.12-1990.*

programmable electronic system (PES). A system, based on one or more computers, connected to sensors and/or actuators on a plant for the purpose of control, protection or monitoring. *IEC 65A(Secretariat)122, August 1991.*

quality assurance. (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured. Contrast with: quality control (1). *IEEE 610.12-1990.*

reliability. The ability of a system or component to perform its required functions under stated conditions for a specified period of time. *IEEE 610.12-1990.* The probability of failure-free operation of a computer program for a specified time in a specified environment (Musa, Iannino, and Okumoto, 1988).

requirements phase. The period of time in the software life cycle during which the requirements for a software product are defined and documented. *IEEE 610.12-1990.*

retirement phase. The period of time in the software life cycle during which support for a software product is terminated. *IEEE 610.12-1990.*

risk. (1) A measure that combines both the likelihood that a system hazard will cause an accident and the severity of that accident. *IEEE P1228 Draft E, July 1991.* (2) The expected detriment per unit time to a person or population from a given cause. *IEEE 577-1976.* (3) The combination of the frequency, or probability, and the consequence of a specified hazardous event. The concept of risk always has two

elements; the frequency or probability with which a hazard occurs and the consequences of the hazardous event. *IEC 65A(Secretariat)122, August 1991*.

safety. The expectation that a system does not, under defined conditions, lead to a state in which human life, limb, and health, or economics or environment, are endangered. Note: For system safety, all causes of failures which lead to an unsafe state shall be included: hardware failures, software failures, failures due to electrical interference or to human interaction, and failures in the controlled object. Some of these types of failure, in particular random hardware failures, may be quantified using such measures as the failure rate in the dangerous mode of failure or the probability of the protection system failing to operate on demand. System safety also depends on many factors that cannot be quantified but can only be considered qualitatively. *IEC 65A(Secretariat)122, August 1991.*

safety integrity. The likelihood of a Programmable Electric System achieving its safety functions under all stated conditions within a stated period of time. *IEC 65A(Secretariat)122, August 1991.*

safety-related software. (1) Software whose inadvertent response to stimuli, failure to respond when required, response out of sequence, or response in unplanned combination with others can result in an accident. Also, software that is intended to mitigate or to recover from the result of an accident. IEEE P1228 Draft E, July 1991. **Safety-critical software**. (2) Software that ensures that a system does not endanger human life, limb, and health. IEC 65A(Secretariat)122, Aug 1991.

semi-formal method. Procedure-based method for the specification, design, and production of software. Examples include logic or function block diagrams, sequence diagrams, time petri nets, and truth tables. Derived from *IEC 65A(Secretariat)122, Aug 1991.*

software development process. The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code and, sometimes, installing and checking the software for operational use. Note: These activities may overlap or be performed iteratively. *IEEE 610.12-1990.*

software diversity. A software development technique in which two or more functionally identical variants of a program are developed from the same specification by different programmers or programming teams with the intent of providing error detection, increased reliability, additional documentation, or reduced probability that programming or compiler errors will influence the end results. See also: diversity. *IEEE 610.12-1990.*

software life cycle phase. The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and check-out phase, and operation and maintenance phase, and may include a retirement phase. Note: These phases may overlap or be performed iteratively. *IEEE 610.12-1990.* Also referred to as: life cycle, lifecycle, and software lifecycle.

software product. (1) The complete set of computer programs, procedures, associated documentation, and data designated for delivery to a user. *IEEE 610.12-1990.* (2) Any of the individual items in (1). *IEEE 610.12-1990.*

software reliability. The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. *IEEE 982.1-1988.*

structured methodology. A step-by-step method for systematically perceiving and partitioning a problem or system. Some main features of structured methodologies include (1) a logical order of thought that breaks large problems into manageable stages, (2) identification of the total system, including the environment, (3) decomposition of data and functions in the required system, (4) use of checklists, intutive, and pragmatic overhead**.** Examples of structured methodologies include Jackson Systems Development (JSD), Modular Approach to Software Construction (MASCOT), Structured Analysis and Design Technique (SADT), and real-time Yourdon). Derived from *IEC 65A(Secretariat)122, Aug 1991.*

system hazard. A system condition that is a prerequisite to an accident. *IEEE P1228 Draft E, July 1991.*

test phase. The period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether requirements have been satisfied. *IEEE 610.12-1990.*

unit testing. Testing of individual hardware or software units or groups of related units. *IEEE 610.12-1990.*

5.2 Acronyms

AFSCP	Air Force Systems Command Pamphlet
ANS	American Nuclear Society
ASQC	American Society for Quality Control
CCFA	Common-cause failure analysis
CPU	Central processing unit
DoD	Department of Defense
DOE	Department of Energy
EIA	Electronics Industry Association
EPRI	Electric Power Research Institute
FAA	Federal Aviation Administration
FIPS	Federal Information Processing Standards
FMEA	Failure Mode and Effects Analysis
HAZOP	Hazard and Operability
HMSO	Her Majesty's Stationary Office
ICSB	Instrumentation and Control System Branch
IEC	International Electrotechnical Commission
IEE	Institute of Electrical Engineers (British)
IEEE	Institute of Electrical and Electronic Engineers
LLNL	Lawrence Livermore National Laboratory
MID-STD	Military Standard

MoD	Ministry of Defence (British)
NRC	Nuclear Regulatory Commission
NRR	Nuclear Reactor Regulation
NSAC	Nuclear Safety Analysis Center
NUREG	Nuclear Regulatory Guide
SQA	Software Quality Assurance

6. Reliability in Safety-Related Systems

Historically, quantitative measures have been used to evaluate the reliability of hardware components. When the need for evaluating the reliability of software arose, some of the measures were applied to software. The usefulness of these software reliability measures has been discussed in many recent articles and at conferences. Pilsworth (1988), in his overview of MoD Defence Standards 00-55 and 00-56, summarizes the concerns and new approaches:

> The approach to safety in the defense industry has been similar to many other industries where great reliance has been placed on the ability to predict the probability of a given safety-related event occurring. At the system level this method is still viable. However, if some functions are to be implemented in computer software, then this approach is not viable at the lower levels. The failure rate of computer software cannot be predicted with sufficient precision before design commences. Computer software failure modes are virtually impossible to predict. Therefore, the approach to implementing safety functions in computer software places great reliance on the requirements definition, specification, design and development methods employed.

Pilsworth's claim is supported by Musa (1989) and Butler (1991), who contend that high-degree reliability measurements cannot be determined for applications such as control software for nuclear power generating stations, avionics, and weapons control systems.

The applicability of mathematical algorithms and models to measure reliability is currently being discussed in industry publications and at conferences. Musa states that at the present time, reliability growth models are insufficiently precise for ultra-reliable systems. These systems require failure rates an order of magnitude lower than those of the commercial systems, to which such models are more commonly applied. It is generally agreed that reliability measurements, including reliability growth models, are applicable to software products that require only minimal or low reliability (failure rate greater than 10^{-3} failures per hour) (Butler and Finelli, 1991). However, the ability of these algorithms and models to measure reliability values needed for ultra-reliable (failure rate less than 10^{-7} failures per hour) software systems is disputed. For an ultra-reliable software system requiring a software failure rate of
10^{-10} failures per hour, 1,141,550 years of testing may be necessary to produce a single error (Butler and Finelli, 1991).

This indicates that reliability models are insufficient as the only method for assessing software reliability. Other methods are needed to supplement the reliability models. One is the application of quality-enhancing software development and operation processes. It is believed that safe and reliable systems can be produced by using quality processes (Musa, Iannino, and Okumoto, 1987). The principal factors that affect software reliability, such as fault generation, fault removal, and the environment, should be considered first. Fault generation depends primarily on the characteristics of the code created or modified and the software development process. The characteristics of created or modified code include size and complexity, and the characteristics of the development process involve

software engineering technologies, tools used, and experience level of personnel. Musa says that fault removal depends on time, operational profile, and the quality of the repair activity. The environment may be modeled with an operational profile that can be used to test the system for detecting faults. Because some foregoing factors are probabilistic in nature and operate over time, software reliability models are generally random processes (Musa, Ianinno, and Okumoto, 1987).

7. Software Life Cycle Practices

Safety-related software development methods and software reliability techniques are performed throughout all phases of the software life cycle. The domestic and international standards reviewed in this report recommend several processes that span multiple life cycle phases. These processes may improve the quality of the software by reducing the number of software faults, thus having a positive influence on the software's reliability. This section describes these processes. Appendix A maps these processes and the information in Section 10 to the individual standards where they are recommended.

7.1 Quality Planning and Procedures

The standards DoD 2168, FAA 013B, 016A, 018A, IEC 65A(Secretariat)122, IEEE 730, and MoD 00-55 detail quality assurance plans for building safety and reliability into a software system. DoD 2167A and IEC 880 expand beyond specifying a plan and include detailed procedures to be performed in software development and maintenance. IEEE 279 is the only standard that does not specifically identify a quality assurance plan or software development process but states that "quality levels should be achieved through the specification of requirements known to promote high quality."

IEC 65A(Secretariat)122, IEEE 730, and FAA 018A specify techniques, methods, and tools that software developers may use to implement quality assurance. Other items addressed in the plans are coding standards (IEC 65A(Secretariat)122, IEEE 730 and P1228, IEC 880, and DoD 2167A), documentation standards (IEEE 730), and software metrics (IEEE 730).

7.2 Software Development Processes

The standards listed in Section 7.1 recommend a common set of software development processes, including requirements definition, design, testing, configuration control, error reporting and corrective action, and verification and validation. DoD 2167A and IEEE 730 extend this set to include risk management of technical, cost, and schedule issues. Some standards specifically list the documentation produced by these processes, whereas others say only that good documentation should be provided.

7.3 Monitoring Software and Hardware

To achieve increased overall system reliability, IEC 65A(Secretariat)122 and IEC 880 both suggest that software supervise itself along with the hardware. These supervision tasks include monitoring the parts of the memory that contain code or static data for changes. Requirements to enable the software to monitor both the hardware and software are included in the IEC 65A(Secretariat)122 software requirements specification. The monitoring functions generally are executed on a separate cpu. Monitoring software can inadvertently become a cause of system failures.

7.4 Design Approaches

IEC 65A(Secretariat)122 and IEC 880 recommend design approaches that can be used when developing a software product. Some of these approaches are defense-in-depth, graceful degradation, redundancy, and hardware and software functional diversity. These approaches are implemented by techniques described in Section 9. MoD 00-55 does not address these fault-detection or fault-tolerant approaches but does recommend adding software code to check data values at boundaries between modules or components. Appropriate action must then be taken if anomalies are found.

7.5 Software Module Design Quality

The complexity of a software product's architecture can increase the probability of the occurrence of a fault. IEC 65A(Secretariat)122 requires design methods that facilitate abstraction, modularity, and other features to control complexity. Modularity, information hiding, and data encapsulation can facilitate software maintenance (IEC 65A [Secretariat]122, 1989). Measuring the number of entry and exit points can identify components that do not properly encapsulate their data and are overly complex. These components are likely to result in high operating system overhead and may be difficult to test completely.

Minimizing the number of entry and exit points in a component and limiting the number of functions in a software component are recognized in the software industry as key features in good structured design and programming. It is recommended that each software component have one entry and one exit point for each major function and that the number of functions per software component be limited to five (IEEE 982.2, 1988). It may sometimes be necessary to increase the number of exit points to allow for error handling.

7.6 Reviews and Audits

Review and audit processes are highly effective in software fault detection and correction. Detecting software faults earlier in the life cycle phases, prior to system integration testing, reduces the cost of removing them. Several standards (DoD 1521B, DOE 5481.1B, and IEEE 1028) are specifically devoted to reviews or audits, and most of the software quality assurance standards address these topics.

All the software quality-assurance planning standards and guidelines that were reviewed for this report require reviews and audits. MIL-STD 882B and MoD 00-55 refer to these reviews as safety reviews. Characteristics of the reviews and audits that were identified in the reviewed standards can be summarized as follows:

- Reviews and audits require items to be measurable against predetermined values. All standards specify that software requirements and hardware and software interfaces must be testable, verifiable, and realizable. In a flow-control application, an example of one such testable requirement might be “Shut off the pumps within 5.0 seconds if the mean water level over the past 4.0 seconds was above 100.0 meters” (Parnas, 1990). In meeting this criterion, requirements and other products from the software development processes can be compared with the expected results during a review or audit.

- Actual timing of the review or audit is specified in IEC 65A(Secretariat)122, IEEE 730, DoD 2167A, and MoD 00-55. Other standards use phrases such as "review after every stage."

- Reviews and audits are subdivided into process and product categories. Process reviews and audits assess the activities performed during the software development. Examples of these reviews and audits are postmortem review, error reporting and correction process review, and managerial review of the quality-assurance plan implementation. Product reviews and audits critique the deliverables of a software process. Examples are requirements review, design review, verification and validation plan review, and physical audit.

The quality of the review or audit process influences the effectiveness of detecting software faults. As software faults are detected and corrected, confidence in the reliability of the software improves. The review and audit processes should be analyzed for their effectiveness in detecting software faults. The time an inspection team prepares for and conducts the review or audit should be compared with the number of software faults detected. The result can then be used as an index (number of hours spent detecting an error) of the effectiveness of the inspection process. Given industry trends, it is more likely that a high value of this index indicates that the reviews and audits processes are not being effectively implemented. However, in an organization with excellent software development processes, the hours spent detecting an error may be high because fewer errors are present, and thus more time may be required to discover any errors.

7.7 Specifying Reliability and Safety Requirements

IEC 880 links software reliability requirements to system requirements by stating that software reliability requirements are an expansion of the system reliability requirements. IEC 880 implies that system reliability requirements should be partitioned among the hardware, software, and firmware components. IEEE P1228 requires that the software requirements specification or the safety plan explicitly include specifications for software to avoid and control safety hazards.

MIL-STD 882B requires system safety design requirements to be specified after a review of pertinent standards, specifications, regulations, design handbooks, and other guidance for applicability to the design of the safety-related system. MIL-STD 882B identifies a separate process specifically for software requirements hazard analysis. Within this process, there are tasks to ensure that system safety design requirements are correctly and completely specified, that they have been properly translated into software safety requirements, and that the software safety requirements will appropriately influence the software design.

7.8 Requirements Checking

Software and system requirements specifications are the foundation for the software end product. If these specifications are not correct, the software may not perform its intended functions. In the best case, the system may then be unacceptable. In the worst case, the system may have hidden software functions or faults that allow an unsafe state to occur. For safety-related software, it is necessary to validate the software and system specifications or requirements. Requirements are traced to the software design and architecture to ensure that no requirement is missing or any unwanted function has been added. The requirements that can be traced to more than one software design or architecture component can be analyzed for possible functionality conflicts. Usually,

tracing the software specifications or requirements is a manual process using matrices, where the software requirements are the rows of the matrix and the design components are the columns. For each requirement, checks are placed in the corresponding design-component cell or cells that satisfy the requirement. Some computer-aided software engineering (CASE) tools may provide assistance with this activity.

7.9 Performance Specifications

The performance of a safety-related system is crucial to its successful operation. Initiation of the various hardware and software components **is** determined by the timely responses of other components. If one component fails to respond at an expected time, an event triggering a fault recovery function may be initiated; or worse, the system could fail, causing an unsafe state to be entered. IEC 880 requires a separate software performance specification, whereas IEEE 730 and DoD 2167A include performance issues with the software requirements.

7.10 Formal Methods

IEC 65A(Secretariat)122 and MoD 00-55 link formal methods to a quality development process by requiring that formal mathematical methods be used in specifying and designing the software. At the highest software safety integrity level, level 4, IEC 65A(Secretariat)122 highly recommends the use of formal methods, semi-formal methods, and/or a structured methodology. MoD 00-55 also requires that the software code be analyzed by a static analysis tool. LLNL's Formal Methods report, *Formal Methods in the Development of Safety Critical Software Systems*, UCRL-ID-109416, addresses this topic in detail.

7.11 Programming Languages

Programming languages may affect the safe functioning of a software system. For example, the programming language chosen and its implementation can affect the results of integer and floating-point arithmetic. Language constructs can allow unsafe practices to be coded into the software product, thus increasing risk. A programming language can allow a software program to jump to an arbitrary memory location or even overwrite an arbitrary memory location. Some programming languages specifically disallow these constructs. Some languages require strong data type definition and perform type checking to avoid the misuse of program variables and also to prevent running out of memory at run time.

IEC 65A(Secretariat)122 and 880 make similar recommendations on the use of programming languages. A programming language should support strong typing of variables, structured programming methods, run-time type and array bound checking, parameter checking, and means to verify the source code with a minimum of effort. IEC 65A(Secretariat)122 contains a table of current programming languages and the recommended usage of these languages at the various software safety integrity levels (level 4 being the highest). ADA, Modula-2, PASCAL, and FORTRAN 77 are highly recommended for software safety integrity levels 3 and 4 only if a subset of the language is used. However, Cullyer, Goodenough, and Wichmann (1991) do not recommend even a subset of FORTRAN 77 for safety-related systems. Cullyer et al. further recommend that at least static code analysis, and preferably formal proofs, be used in conjunction with the subsets of ADA, Modula-2, and PASCAL. No recommendation is provided for use of an assembly language for either of these software safety integrity levels. C, PL/M, and

BASIC are not recommended by IEC 65A(Secretariat)122 for implementing software at the software safety integrity levels 3 or 4.

7.12 Complexity and Scalability

Complexity metrics can provide very useful design guidance and indications of product quality when they are used as part of a well-managed software development effort. However, using these metrics to predict safe performance has very little support in experience. IEEE 982.1 recommends several design-complexity metrics, including Software Science Measures, Graph-theoretic Complexity for Architecture, Cyclomatic Complexity, Data or Information Flow Complexity, and Design Structure.

Closely related to complexity is scalability, changing the size of a real-time system. Scaling a real-time system can have unexpected effects on system response, reliability, and correctness. Presently, many issues in scalability are under active study, although some results (such as performance analysis techniques) are being used in practice. LLNL's work in progress report *Real-time Systems Complexity and Scalability*, Draft 2, November 1991, addresses these issues in more detail.

8. Safety Concepts

Safety is a system property. Achieving and maintaining safety involves all aspects of a system, including human, electronic, and mechanical components. Many standards do not clearly distinguish between software reliability and safety. Examples of this are IEC 65A(Secretariat)122, IEC 880, MoD 00-55 and DoD 2167A, which discuss safety as a part of their overall software quality requirements. Other standards, such as IEEE P1228, MIL-STD 882B, and MoD 00-56, are devoted specifically to safety. Appendix B includes a table that maps the information provided in this section to the individual standards.

*T*his report identifies software reliability as the probability that the software will not cause the failure of a system for a specified time under specified conditions (IEEE 982.1, 1988). Safety-enhancing practices ensure that the system will operate without creating a system hazard. The essential issue is that a highly reliable system can have very infrequent but catastrophic failure modes. On the other hand, a safety-related system may "fail-safe," shutting down the system it is controlling to prevent the system from proceeding further into a hazardous state. The reliability of such a system may be considered low because the system has failed to continue to operate.

In a fail-safe system with two computers operating in active redundancy, the outputs of the two are compared. When they disagree, it is assumed that one of the computer systems has failed. If a high degree of safety is required, both computers need to be shut down and the system brought to a safe fall-back state. This is necessary since the system cannot safely be controlled without redundancy. However, if a high degree of reliability is required, and three or more computer systems are used for redundancy, there is a possibility that the faulty computer can be identified and the operation continued without any redundancy.

8.1 Safety Plans

Each of the standards, IEEE P1228, MIL-STD 882B, and MoD 00-55, requires a safety plan that is part of a safety program. This plan should be updated throughout the software life cycle phases. The safety plan includes management and technical procedures and practices to be performed in developing a safety-related system. The procedures and

practices described in a safety plan include: (1) organization and structure of the safety program, including functional relationships and lines of communication, (2) review and approval procedures for the safety-related tasks, (3) personnel qualifications for safety-related tasks, (4) recording results of the safety-related tasks such as a safety records log containing the results of hazard analyses and their resolutions, (5) configuration management activities, (6) staffing and funding requirements, (7) training requirements for operation and maintenance personnel, and 8) risk assessment and hazard analyses procedures.

8.2 Hazard Severity and Software Safety Integrity Levels

The degree to which software safety techniques and processes are applied may have a direct correlation with the severity of the hazard when the system fails. Two similar concepts, hazard severity level and software safety integrity level, can be used to determine the necessity for applying specific safety-related processes. Many standards, including IEEE 352, 577 and P1228, recommend or require the categorization of failures but do not associate these levels with specific techniques and processes. MoD 00-55 and 00-56 map the categorization to specific processes and documentation. MIL-STD 882B and IEC 65A(Secretariat)122 extend this categorization to specific techniques within the processes. The IEEE 1012 Software Verification and Validation Plan Standard working group is investigating a similar concept to be included in the standard's next revision.

The hazard severity level, as described by MIL-STD 882B and MoD 00-56, uses four categories (catastrophic, critical, marginal, and negligible) to provide guidance in determining the techniques and processes to be implemented in the development of a safety-related system. Both standards also contend that the probability of the occurrence of a hazard must be considered in determining the techniques and processes to be implemented. MIL-STD 882B uses five probability levels ranging from frequent to improbable—in contrast to MoD 00-56's six levels, which include an additional level termed “incredible.”

The concept of software safety integrity levels is used in IEC 65A(Secretariat)122. These levels are determined by the level of risk associated with the software. The consequences of the loss of human life, injury or illness to humans, environmental pollution, and loss of or damage to property are considered when determining the software safety integrity level. IEC 65A(Secretariat)122 identifies five levels (level 4 down to level 0) of software safety integrity: very high, high, medium, low, and non-safety-related. These levels map to degrees of recommendation (highly recommended, recommended, no recommendation, and not recommended) applied to various techniques and processes that the standards identify. This report describes some of those recommendations in the discussions of techniques in Section 9.

8.3 Safety Analysis

One method of performing system safety analysis is failure mode and effects analysis (FMEA). IEEE 352 and 577 discuss FMEA extensively. IEEE 352 classifies the FMEA as a reliability analysis. The goal of a FMEA is to identify the modes of system failure and their consequences. Often the FMEA is extended to include common-mode failures of redundant components. This extension is called common-cause failure analysis (CCFA). CCFA lists the system failures that could occur in all operational modes (automatic, manual, test, and bypass). Additional common-mode failures can be identified by reviewing the system boundary conditions and the customer’s view as to what is a failure. Environmental effects such as fires, earthquakes, floods, and electromagentic interference

also should be considered. FMEA and CCFA are system-wide safety methods. They will not be discussed in depth in this report. However, techniques performed during a FMEA and CCFA that are applicable to assessing software reliability are discussed in Section 9.

Another method of safety analysis is a software Hazard and Operability Study (HAZOP). HAZOP covers all phases of the life cycle. A team of engineers (computer, instrument, electrical, process, safety, and operational) headed by a trained hazard analyst examine the software system and its operation for environmental failures that may lead to a hazardous situation. The HAZOP differs from a FMEA in that the focus of a FMEA is on the failure modes of components of the system, whereas a HAZOP analyzes the functions of the components in the event of a fire, flood, earthquake, explosion, or toxic release. IEC 65A(Secretariat)122 highly recommends FMEA, CCFA, and HAZOP for meeting very high software safety integrity requirements. MIL-STD 882B's and MoD 00-56's primary focus is HAZOP analysis.

MIL-STD 882B and MoD 00-56 require similar hazard analyses. However, MoD 00-56 groups several hazard analysis processes into a single, system hazard analysis and uses different terminology. In the concept phase of the system life cycle, a preliminary hazard analysis is performed to assess the risk of the concept or system. A hazard analysis is then performed on each subsystem to identify all components (hardware and software) whose performance, performance degradation, functional failure, or inadvertent functioning could result in a hazard. Other hazard analyses include system, operating and support, and occupational health hazard analyses. Though the standards do not specify the timing of either these hazard analyses or the subsystem hazard analysis, details of system design may be needed before the analyses are performed. The system hazard analysis identifies hazards and assesses the risk of the total system design, including the subsystem interfaces. The operating and support hazard analysis identifies and evaluates hazards resulting from the implementation of operations or tasks performed by humans. And lastly, the occupational health hazard analysis identifies health hazards and recommends engineering controls, equipment, and/or protective procedures to reduce the associated risk to an acceptable level. These health hazards include toxic materials, noise, heat stress, ionizing radiation, ventilation, and radiation barriers.

8.3.1 Software Specific Safety Analysis

MIL-STD 882B and MoD 00-56 have specific sections related to software system safety hazard analyses. MoD 00-56 specifies a single software functional hazard analysis while MIL-STD 882B identifies six hazard and safety analyses for the software. Four of the six MIL-STD 882B hazard analyses (software requirements, top-level design, detailed design, and code-level software hazard analyses) use the deliverables from their associated tasks in DoD 2167A as the basis for performing their analyses.

In addition to the software-specific hazard and safety analysis, MIL-STD 882B recommends an extensive set of overall system hazard and safety analyses that can be applied to hardware and/or software. However, MIL-STD 882B recommends applying the software-specific section of the standard to large or complex software products. For smaller-scale software products, the overall system hazard and safety analyses can be modified to include software hazard analysis requirements.

9. Software Reliability Processes and Measurements

Reliability in general can be thought of from two different perspectives. The customer may consider a system reliable if it has a high availability (high mean-time-to-failure),

whereas the software developer's system designers may consider a system reliable if it has a low failure rate (failures per thousand lines of code). Techniques for both perspectives are included in this section. A further discussion in Section 9.1.1 describes the differences.

Reliability analysis uses techniques, procedures that implement those techniques, and estimation or prediction measurements throughout all phases of the software life cycle. The techniques use current trends either to estimate system or software reliability at the current point in time or to predict its value at a future time. A technique performed in one phase of the software life cycle may determine what technique, if any, should be performed in another phase. Some software reliability techniques are closely associated with techniques for assessing overall system reliability (e.g., modeling fault trees and categorizing failures). In those instances, we shall discuss the appropriate system reliability technique.

Several standards and publications use the software development life cycle phases as a base structure for discussing software reliability techniques. In Section 8 of this report, these phases also provide a structure for estimating software reliability. Software development standards and literature identify the software life cycle phases differently. To avoid confusion, this report defines the software life cycle phases as (1) concept, (2) requirements, (3) design, (4) implementation, (5) test, (6) installation and verification, (7) operation and maintenance, and (8) retirement. Descriptions of these phases are found in Section 3.1, Definitions. As software reliability does not apply in the retirement phase of the software life cycle , we do not include it. Appendix C contains a table mapping the various techniques and procedures in this section to the individual standards.

Several techniques can be applied to more than one software life cycle phase, and some overlap multiple phases. Some techniques are complementary, and some may be substituted for another technique. The strengths, weaknesses, and related measures of each technique are discussed in the present section. Each measure is identified by an (I) for indicator or a (P) for predictor.

Related measures or metrics discussed are from IEEE 982.1, Standard Dictionary of Measures to Produce Reliable Software, and from AFSCP 800-14, Software Quality Indicators. AFSCP 800-14 identifies seven measures that are a subset of the 39 IEEE 982.1 measures. More than one measure may be appropriate for a specific technique. Some measures can replace others, and some are best used to complement each other. Which measures to apply may best be decided by the actual personnel using the technique. Not all the measures listed may be appropriate to use or may be considered useful by some software reliability and safety-related software experts. The Considerations, Training, Experience, and References sections in IEEE 982.2 should be carefully reviewed before any of these measures are applied.

The IEEE 982.1 and its guide-to-use standard, IEEE 982.2, include a description of each measure's use, the input parameters for the measure, an interpretation, problems to consider in using the measure, the background needed to perform the measurement, an example, the benefits of the measure, user experience, and references. AFSCP 800-14 uses a similar format that is modified for application in the U.S. Air Force environment. Appendix D contains a brief description of each measure. The measures are grouped in two categories—product and process—and nine subcategories: (1) errors, faults, and failures counting; (2) mean-time to failure and failure rate; (3) reliability growth and projection; (4) remaining faults estimation; (5) completeness and consistency; (6) complexity; (7) management control; (8) coverage; and (9) risk, benefit, and cost

evaluation. Table 9.1 contains a list of these reliability measures. (Table 9.1 is reproduced from IEEE 982.2 with the permission of the IEEE.)

9.1 Concept Phase

9.1.1 Set Reliability Goals

Software reliability goals are established by the software development staff working closely with the systems engineers and end-user representatives. In developing a software product, the customer's perspective on reliability must be considered when establishing these goals. Usually, software reliability goals and their objectives are specified from the software designer's viewpoint (faults per thousand lines of delivered source code). Using the customer's viewpoint (failures per thousand CPU hours) provides a better measure for meeting the reliability goals (Musa and Everett, 1990). Goals should consider the overall system objectives, system performance requirements, rate of demand on the specific safety system, complexity of the system design, consequences of safety system failure, testing limitations, the customer's requirements for the system, and in the case of nuclear power generating systems, any regulatory requirements. Goals must be weighed against their impact on the environment and human safety, their importance to the total system performance, and the practical feasibility of achieving them. Reliability goals should be included as an explicit part of the requirements specification. Software should be evaluated according to its ability to meet the reliability goal's objectives; therefore, the objectives must be testable.

The amount of reliability testing to be performed should be carefully considered because testing takes time and increases costs (Everett, 1990). Planning for reliability goal testing should be performed in the requirements phase, as suggested by Everett (1990).

Benefits: Establishing reliability goals during the concept phase focuses the development team **on** those goals (Musa and Everett, 1990).

Deficiencies: Early in the software life cycle phase, it is sometimes difficult to obtain enough information to specify meaningful reliability goals.

Interrelationship with other capabilities: Once the goals for reliability are determined, procedures to achieve these goals and tests to verify the reliability must be established.

Addressed by: IEEE 352, 577, and 982.2.

Related Metric: RELY-Required Software Reliability (I).

9.1.2 Identify and Categorize Failure Modes

Identifying and categorizing system failure modes are major tasks within a FMEA or a HAZOP. This technique is applicable to an entire system. Separating software failure modes from those of the entire system is difficult. Once the failure modes have been listed, they can be categorized by their severity and probability of occurrence as they relate to safety and reliability. Understanding how these concepts apply to the failure categories helps identify the differences between safety and reliability.

In viewing reliability without considering safety, failure modes can be categorized by the frequency with which they are expected to occur. Minimizing the frequency of failures will increase reliability but may not improve safety because the failures that occur most

often may not be those that have the greatest impact on human life or on the environment. Severity of consequences must be considered by the software developer when categorizing failure modes. From a safety standpoint, failure modes can be categorized for their contribution to hazardous situations. Identifying the consequences of the failure can help in the categorization.

Benefits: Categorization of failure modes can determine the priority that should be given to prevent failure. Once critical failure modes are identified, test procedures and other reliability processes can be tailored to prevent the occurrence of the most risky failure modes (in terms of safety) and/or the most frequent failure modes (in terms of reliability).

Deficiencies: The skills of the individuals performing each analysis will affect the completeness and thoroughness of the application of this technique. Even when this task is performed by persons highly qualified in determining the failure points, there is no assurance the all possible failure modes have been identified.

Table 9.1. Measure classification matrix. Source: [19]. (1) AFSCP 800-14 measure.

	Product measures						Process measures		
Measures (experience)	Errors, faults, failures	Mean time to failure, failure rate	Reliability growth and projection	Remaining product faults	Completeness and consistency	Complexity	Management control	Coverage	Risk, benefit, cost estimation
1. Fault density (1)	•								
2. Defect density (1)	•								
3. Cumulative failure profile	•								
4. Fault-days number	•						•		
5. Functional or modular test coverage					•			•	•
6. Cause and effect graphing					•			•	
7. Requirements traceability	•				•			•	
8. Defect indices	•						•		
9. Error distributions							•		
10. Software maturity index			•						•
11. Man-hours per major defect detected							•		•
12. Number of conflicting requirements	•				•			•	
13. Number of entries/exits per module					•	•			
14. Software science measures				•		•			
15. Graph-theoretic complexity for architecture						•			
16. Cyclomatic complexity					•	•			
17. Minimal unit test case determination					•	•			
18. Run reliability			•						
19. Design structure (1)						•			

20. Mean time to discover the next *K* faults									•
21. Software purity level			•						
22. Estimated number of faults remaining				•					
23. Requirements compliance	•				•			•	
24. Test coverage (1)					•			•	
25. Data or information flow complexity						•			
26. Reliability growth function			•						
27. Residual fault count									
28. Failure analysis using elapsed time			•	•					
29. Testing sufficiency (1)			•	•				•	
30. Mean-time-to-failure		•	•						
31. Failure rate		•							
32. Software documentation and source listings (1)					•				
33. RELY (required software reliability)								•	•
34. Software release readiness									•
35. Completeness (1)					•				
36. Test accuracy				•	•			•	
37. System performance reliability			•						
38. Independent process reliability			•						
39. Combined hardware/software system operational availability			•						

Interrelationship with other capabilities: This categorized list can be used by any future software life cycle process to focus on the most important aspects of safety and/or software reliability. Fault-tree modeling, cause-consequence diagrams, system functional diagrams, and reliability block diagrams can be used to categorize and list the failures.

Addressed by: IEEE 352, 577 and P1228, MIL-STD 882B, and MoD 00-55 and 00-56.

Related Metrics: No metrics were found that apply to this technique.

9.1.3 Fault-tree Modeling

Fault-tree modeling uses logical, graphical tree-like structures to illustrate the system and to identify failure dependencies and propagations. Fault-tree modeling relates component fault to system failures. This modeling technique uses Boolean algebra to predict system reliability or availability. It also can be applied to software reliability. Fault-tree modeling is frequently performed after a FMEA or CCFA to diagram and communicate information developed during those analyses. Fault-tree modeling represents the system in terms of events leading to failures. However, the model determines the probability of a single event and thus cannot be used to analyze multiple events. The highest-level undesired event is specified. The system failure logic is then traced down to the level that shows the lowest-level component fault that can cause the event. Some computer tool packages are available to use with this modeling technique.

Benefits: Fault trees force the software developer or systems analyst to actively identify the possible failures and events showing dependencies, common-mode failures, and sequences of events. Fault trees can identify critical aspects of system behavior. The modeling process also may show where redundant hardware is needed. Fault-tree modeling allows the user to trace through several interconnected systems to find the root causes of the top event.

Deficiencies: Fault trees are seldom produced with enough detail to discover subtle common modes, and they do not handle multiple system states. Detailed fault trees may be unmanageably large, and computer time can become excessive. Therefore the user must determine when and how to terminate the analysis. Further, since a separate fault tree is required for each top event, the user must carefully define the top event if the analysis is to be beneficial.

Interrelationship with other capabilities: Fault trees are useful after failure modes have been identified and categorized, and they may uncover additional failure modes that have not been identified. The logical failure paths and probabilities that fault-tree modeling produces also can be used in Monte Carlo modeling (Section 9.2.3).

Addressed by: IEC 65A(Secretariat)122, IEEE 352 and 577, and MoD 00-56.

Related Metric: Cause and Effect Graphing (I).

9.1.4 Event-tree Analysis

Event-tree analysis builds a graphical model that indicates the sequence of events that develop after an event has occurred. The consequences of primary and secondary events can then be evaluated for their impact on the safety of the system. An event tree begins with a fault (a cause) and proceeds forward through all of its consequences. Starting from an initiating event, a line is drawn to the first conditional sequence. Branches labeled “yes” and “no” are used for alternative paths from this event to the next condition. This procedure continues until all conditions from the initiating event have been drawn. The event tree can be used to determine the probability of the consequences of the initiating event.

Benefits: Event-tree analysis has several major advantages. The tree is easy to draw and to understand once the sequence of events has been established, and probabilities can be derived from the analysis.

Deficiencies: It is sometimes difficult to identify a complete sequence of conditions and to account for the various failure modes. It is also difficult to take into account dependent failures, common equipment, and common fault interactions. Another problem is that an event tree can become very large.

Interrelationship with other capabilities: The event tree differs from the fault tree in that the fault tree starts with a failure (a result or consequence of an event) and works backward through all the causes to determine the root cause, whereas the event tree starts with a failure and works forward to determine all of the possible consequences. Event trees complement fault trees and are a subset of the cause-consequence diagram.

Addressed by: IEC 65A(Secretariat)122.

Related Metric: Cause and Effect Graphing (I).

9.1.5 Cause-consequence Diagram

The cause-consequence diagram combines fault-tree and event-tree analyses to produce a graph that describes the conditions for propagation of an event. These diagrams are used to calculate the probability of occurrence of critical consequences. The diagram is generated by starting from a critical event, then tracing the cause of the event backward (fault-tree modeling) and the consequences forward (event-tree analysis). Time delays are included. To simplify the diagram, logical symbols can be used for the event propagation lines.

Benefits: Since the cause-consequence diagram is a combination of fault-tree and event-tree analysis, it inherits the benefits of both those techniques. In addition, the cause-consequence diagram provides a view of the cause of the failure and its consequences for individual components and for the entire software product.

Deficiencies: Because this technique combines fault-tree modeling and event-tree analysis, it increases the number of events that are modeled. This produces even larger—and possibly unmanageable—diagrams.

Interrelationship with other capabilities: Cause-consequence diagrams are a combination of fault trees and event trees.

Addressed by: IEC 65A(Secretariat)122.

Related Metric: Cause and Effect Graphing (I).

9.1.6 Reliability Block Diagram

A reliability block diagram compares closely with a system functional diagram, illustrating the normal functioning of the system and the interrelationship of events. These diagrams describe the system in terms of events leading to success. First the system boundaries and initial conditions are defined. Then, if a FMEA or functional diagram has been developed it can be used to define the blocks in the diagram. Each block in the diagram represents a component of the system, and the components are grouped into sets. Paths between components are serial if the failure of a single component in the set leads to the failure of the system (Figure 9.1). If component A is successful, the system will remain operational and component B will determine the success or failure of the system. If B is successful, component C will determine the success of failure. If either A, B, or C fails, the system fails.

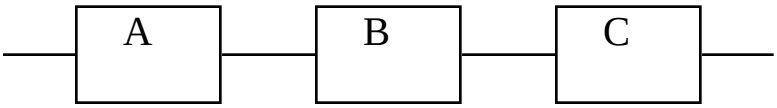

Figure 9.1. Single-failure reliability block diagram.

If redundant components are used, the paths are parallel, creating an n-out-of-m system logic structure, where n = the number of redundant components that must be successful for the system not to fail and m = the total number of components (Figure 9.2). The system remains operational if any two of the three components are successful. This is described as a 2-out-of-3 system logic structure. For example, if components A and B are successful but C is not, the system remains operational.

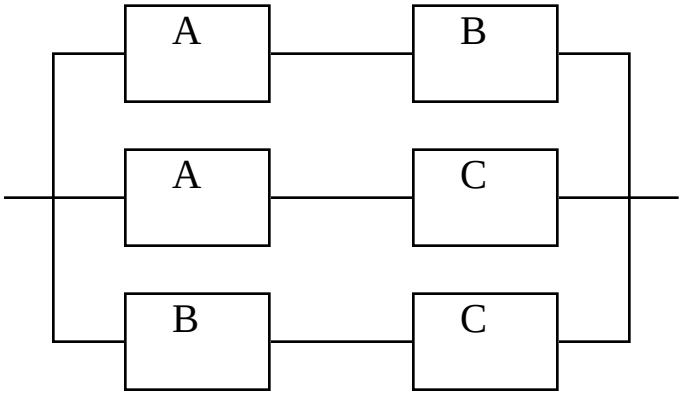

Figure 9.2. Two-out-of-three logic reliability block diagram.

Benefits: The logic of a reliability block diagram is relatively simple, and failure rates can be calculated easily (Lawrence, 1991). Reliability block diagrams may indicate that redundant components must be added to achieve the desired reliability.

Deficiencies: The reliability block diagram assumes that the failure of a component is statistically independent and that the failure rate is constant.

Interrelationship with other capabilities: Reliability block diagrams are useful after failure modes have been identified and categorized. They can be used in FMEAs. A fault tree identifies possible failures and events leading to failures, whereas a reliability block diagram describes system events that lead to success.

Addressed by: IEC 65A(Secretariat)122 and IEEE 352 and 577.

Related Metric: Cause and Effect Graphing (I).

9.1.7 Probability Modeling

Probability formulas are used with truth tables, Venn diagrams, and Boolean algebra to estimate the reliability or availability of a system. These simple formulas use the sum of the component probabilities to calculate system-wide availability or reliability . Availability is calculated from the probability of system success. The probability of success can be determined by using a reliability block diagram (Figure 9.2) to create a truth table (Table 9.1).

Table 9.1. Truth table for Figure 9.2.*

A	B	C	System	Boolean term
0	0	0	0	$\bar{A}\bar{B}\bar{C}$
0	0	1	0	$\bar{A}\bar{B}\ C$
0	1	0	0	$\bar{A}\ B\bar{C}$
0	1	1	1	$\bar{A}\ BC$
1	0	0	0	$A\bar{B}\bar{C}$
1	0	1	1	$A\bar{B}\ C$
1	1	0	1	$AB\bar{C}$
1	1	1	1	ABC

*0 = failure; 1 = success.

Then the success probabilities are summed. This technique can be used to model software diversity in a computer system, where A, B, and C are diverse software components. The probability modeling equation for availability is

$$\text{availability} = P(\bar{A}\ BC) + P(A\bar{B}\ C) + P(AB\bar{C}\ ) + P(ABC)$$

where the success of component A is denoted by A and $\bar{A}$ denotes its complement, the failure of A. Reliability is calculated by using the failure aspects of the model to create the probability of failure. Thus, reliability = 1 – probability(failure). The probability modeling equation for reliability is

$$\text{reliability} = 1 - (P(\bar{A}\bar{B}\bar{C}\ ) + P(\bar{A}\bar{B}\ C) + P(A\bar{B}\bar{C}\ ) + P(\bar{A}\ B\bar{C}\ )).$$

For a well-defined system, the number of ways to show failure is less than the number of success paths. Thus, it is easier to calculate reliability using Boolean algebra for the probability of failure than to calculate availability.

Benefits: Probability modeling is based on techniques previously applied, such as reliability block diagrams or fault-trees. If these techniques have been used, performing the modeling techniques is straightforward. The use of fault-trees leads naturally to Venn diagrams and Boolean algebra.

Deficiencies: If truth tables are used on large problems, the number of terms produced becomes unmanageable, and the technique becomes cumbersome. Boolean techniques do not adequately handle multiple states.

Interrelationship with other capabilities: Probability modeling uses fault trees and reliability block diagrams to determine the elements in their probability equations.

Addressed by: IEC 65A(Secretariat)122 and IEEE 352 and 577.

Related Metric: Cause and Effect Graphing (I).

9.2 Requirements Analysis Phase

9.2.1 Operational Profile Analysis

An operational profile describes how a computer system will operate in the specified environment. It is developed to reflect how the end user will use the product. System inputs and their frequency of occurrence are identified from previous versions of the software, similar software products, and estimates for new features. From this information, a profile of computer system usage can be developed. As an example, a profile may show that during a nuclear power plant start-up, the plant's nuclear steam supply system instrumentation and control system receives two operator requests for data and one operator-generated command per minute on an average. During a loss-of-coolant accident, the same system will receive a maximum of fifteen requests for data and five operator-generated commands per minute. This profile can be used to design user interface functions, input and output screens, and test cases. For systems that are initiated by an operator, the sequence of inputs is considered a single run of the software product. The operational profile is actually a set of relative frequencies of occurrence of the run, usually expressed as fractions of the total set of runs (Musa, Iannino, and Okumoto, 1987). If the runs are independent, it is fairly easy to determine the probabilities of occurrence. However, when runs depend on one or more previous runs, determining the probabilities is more complicated.

Equivalence classes, sets of inputs that cause the system to enter the same state, can be generated from the operational profile. Equivalence classes use a minimum of test data to increase the adequacy of software testing. Input data are partitioned based on their equivalence relationship. Then test cases are developed to cover each partition. Equivalence classes may be defined either from the specification or from the internal structure of the software. IEC 65A(Secretariat)122 addresses the use of equivalence classes in an operational profile. This technique also is discussed by Musa (1989), Musa, Iannino, and Okumoto (1987), Everett (1990), and Musa and Everett (1990). For software with very high software safety integrity levels, IEC 65A(Secretariat)122 highly recommends the use of equivalence classes in testing.

Benefits: The profile can help increase productivity and reduce costs during the design and implementation phase by helping to guide the focus of development resources (Musa and Everett, 1990). The profile can be used to generate a user's manual for the system. Information on frequency of use can suggest simpler software designs. For example, a developer may find that a simple manual recovery design, rather than a complex automated recovery approach, is adequate for an operating condition that occurs infrequently (Everett, 1990).

Deficiencies: In nuclear reactor systems and similar safety-related systems, there may be many diverse sets of inputs that cause the system to reach the same state. It may therefore be impractical to identify input sequences and their probability of occurrence. The use of equivalence classes for testing may minimize this deficiency.

Interrelationship with other capabilities: At this time, no relationships with other techniques have been identified.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.2.2 Markov Modeling

Markov modeling is a mathematical tool used to calculate the reliability or availability of a system. The Markov model uses the concept of the state of a system and transitions among its various states. A system is in a particular state when it satisfies all conditions of that state; it passes from one state to another when a particular event occurs. Such events are called transitions. For reliability modeling, each state represents

a distinct combination of working or failed modules or components; there is a probability for each transition into a state and each transition out of a state.

A simple example is shown in Figure 9.2.2. Two identical modules having the same probability of failure and repair times are used for redundancy. Repairs cannot occur simultaneously. State 0 exists when both modules are working, State 1 when one module has failed with a failure rate of F and is being repaired with a repair rate of R, and State 2 when both modules have failed and are in repair. Since repairs are not performed on the two failed modules simultaneously, there is no transition out of State 2 into State 0. Then the probability of transition from State 0 to State 1 is $2F$dt and the probability of returning to State 0 is Rdt. The probability of staying in State 0 is $1 - 2F$dt. The additional probabilities for the transitions are shown in Figure 9.2.2. The Markov process uses these transition probabilities to determine a state probability equation for entering or staying in a particular state. These equations can be mathematically solved and reliability can then be estimated.

The two basic types of Markov model are chains and processes. The Markov chain uses matrix multiplication in discrete time to determine the set of probabilities, whereas the Markov process uses differential equations over continuous time. The Markov process is mainly used for estimating software reliability. For the Markov process, the transition out of a state is independent of the transitions that caused the state to be entered. When used to determine software reliability, this assumption is reasonable because software failure analysis depends mainly on the faults remaining and the operational profile (Musa, Iannino, and Okumoto, 1987). IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

Benefits: Any state-tree-based model (event-tree, fault-tree, etc.) can be translated into a Markov model. Markov models can handle complex systems that are difficult to model using fault trees, reliability block diagrams, and probability models. Markov models also can handle redundant systems in which the level of redundancy varies with time due to component failure or repair. A major benefit of Markov models is that hardware components and software modules can be treated in the same model.

Deficiencies: A simple fault tree can produce a Markov model with tens of thousands of states. Personnel performing this modeling must have sophisticated knowledge of mathematics.

Interrelationship with other capabilities: Markov modeling uses the results from fault-tree or reliability block diagram analysis.

Addressed by: IEC 65A(Secretariat)122 and IEEE 352 and 577.

Related Metrics: Independent Process Reliability (I), Combined Hardware and Software Operational Availability (P).

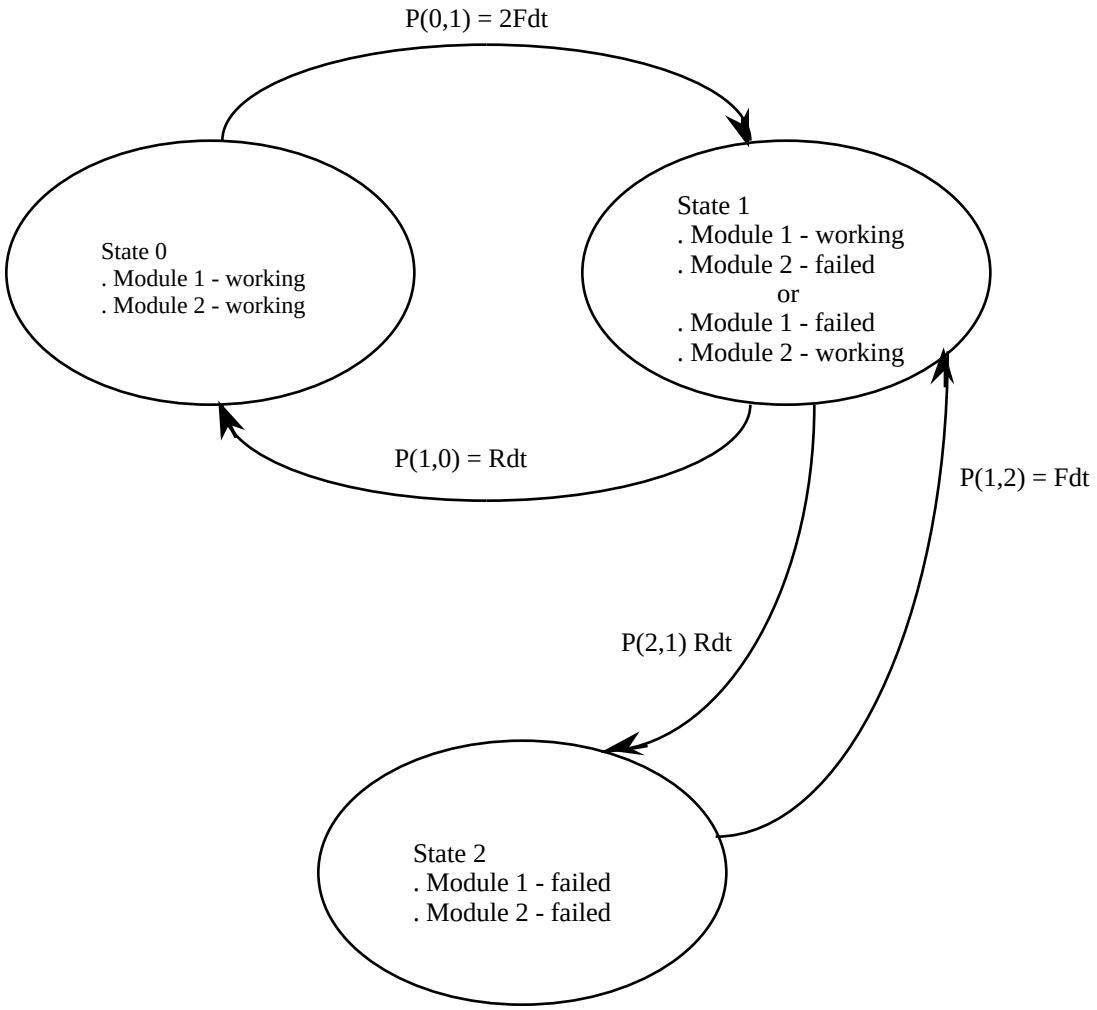

Figure 9.2.2

9.2.3 Monte Carlo Modeling

Monte Carlo modeling uses a random-decision generator, such as a random number generator, to determine the failure of hardware components or software modules. Using computer calculations, the model randomly chooses a component or module to fail. A fault tree or other graphical form is used to relate the success or failure of the system to the success or failure of its components. Thus, the mean time to failure for the system can be determined from the known reliability of its components by conducting repeated trial runs. For each run, the random failure of components is used to determine the success or failure of the system. The average of the times to failure should converge to the mean time to failure if a sufficiently large number of runs are completed. Monte Carlo modeling is frequently used with fault trees to include parameters such as lognormal distribution of time to repair. Since safety probabilities are much smaller than those for reliability, they are less likely to be chosen as a failure by the Monte Carlo random-decision generator. Monte Carlo modeling therefore estimates reliability more accurately than safety.

Benefits: The Monte Carlo model can be applied to a complex system. Statistical variations in daily or weekly model data can be handled with this model. The randomness of the Monte Carlo model may choose combinations of failures that a human may overlook or may be biased against analyzing.

Deficiencies: A large number of computer runs, which may be expensive, are required to show convergence of the mean time to failure. However, as hardware systems increase their computer power and speed, this disadvantage may disappear.

Interrelationship with other capabilities: The Monte Carlo model can be used with fault-tree analysis.

Addressed by: IEC 65A(Secretariat)122 and IEEE 352 and 577.

Related Metrics: Mean Time to Discover the Next K Faults (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P).

9.2.4 Certifying Tools and Translators

This technique has an objective similar to the certification of reused software (discussed in Section 9.5.1). The reliability of a software product is affected by any tool that produces its software components or by the output of any other product that is then used as input to the new software components. Independent organizations use domestic or international standards to certify a tool. Tools in all development activities (specification, design, code, test, and validation) can be evaluated for certification. If a translator has small deficiencies, the specific language constructs affected can be noted and those items avoided during software development. IEC 65A(Secretariat)122 states that immature translators must not be used for safety-related software. The selection of tools and support software should consider the tool's costs, benefits, performance, maturity, usability, interoperability, and maintenance.

Benefits: Level of confidence in a software product increases when tools that produce the software are certified.

Deficiencies: The range of certified tools is limited. Currently, translators (compilers) are the only tools that are regularly certified. The method of certification impacts the quality level of tool certification.

Interrelationship with other capabilities: At this time, no relationships with other techniques have been determined.

Addressed by: IEC 65A(Secretariat)122 and MoD 00-55.

Related Metrics: No metrics were found that apply to this technique**.**

9.3 Design Phase

9.3.1 Determine Factors That Influence Reliability

No standard addresses factors that influence reliability. Musa, Iannino, and Okumoto (1987) and Musa and Everett (1990) discuss how to determine such factors, which they categorize as controllable and uncontrollable. Controllable factors include design inspections, thoroughness of design inspections, and time and resources devoted to testing. Uncontrollable factors include program size, volatility of requirements, and average experience of development staff.

Benefits: Controllable factors can be changed to increase reliability. Awareness of uncontrollable factors can help the developer minimize their impact on reliability.

Deficiencies: The standards and publications reviewed do not provide enough information to address the deficiencies of this technique.

Interrelationship with other capabilities: The information resulting from the use of this technique will identify strong and weak reliability processes and procedures in the software development.

Addressed by: No standards addressed this technique.

Related Metrics: No metrics were found that apply to this technique.

9.3.2 Reliability Time Line Model

No standard addresses the use of a reliability time-line model. However, Musa and Everett (1990) discuss such a technique. A reliability time line can be constructed to indicate the trend of reliability growth over the life of the software product. This time line can be used to evaluate progress toward achieving reliability goals. If the expected progress is not being achieved, project personnel may need to be reassigned, the development process may need to be restructured, or the software design may need to be reworked.

Benefits: Graphing reliability values and monitoring progress toward reliability goals focuses the development team on software reliability procedures and processes.

Deficiencies: The standards and publications reviewed do not provide enough information to address the deficiencies of this technique.

Interrelationship with other capabilities: Graphs can be compared with the reliability growth model to evaluate techniques and processes used during software development.

Addressed by: No standards addressed this technique.

Related Metrics: Cumulative Failure Profile (I), Reliability Growth Function (P), Failure Analysis Using Elapsed Time (P).

9.3.3 Fault Detection and Diagnosis

Fault detection is the process of checking a system for erroneous states caused by a fault. Through the use of diagnostic programs, the software checks itself and hardware for incorrect results. The diagnostic programs can be run periodically or continuously as background processes. Diagnostic programs may include duplicating a calculation two or more times, parity checks, and checksums. From a system perspective, fault detection is based on diversity and on redundancy of hardware and software components. Voting is used to decide the correctness of the redundant components. Fault detection checks values and timing for faults that may be physical (e.g., temperature and voltage instruments), logical (e.g., error-detecting codes), functional (e.g., assertions), or external (e.g., feasibility checks). Fault detection techniques can identify "safe states" where the system is operating properly. As time progresses, the software may detect a state that is not safe. The software can then return the system to the previously known safe state, thus increasing the safety of the overall system. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

Benefits: Small subsystems are identified during fault detection techniques. These smaller subsystems allow a more detailed diagnosis of possible system faults.

Deficiencies: Significant amounts of automated fault detection and diagnosis can slow down the real-time performance of a system and add to its complexity.

Interrelationship with other capabilities: The safety bag (Section 9.3.4) is a form of this technique.

Addressed by: IEC 65A(Secretariat)122 and 880.

Related Metrics: No metrics were found that apply to this technique.

9.3.4 Safety Bag

In this technique, an external monitor, called a safety bag, is implemented on an independent computer using a different specification. The primary function of the safety bag is to ensure that the main system performs safe—but not necessarily correct—operations. The safety bag continually monitors the main system to prevent it from entering an unsafe state. If a hazardous state does occur, the system is brought back to a safe state by either the safety bag or the main system.

Benefits: Safety-bag software is developed from specifications that are different from those for the main computer system, thus reducing the risk of common-mode failures. Safety-bag software is normally simpler than the software in the main computer system, hence reducing the cost of development compared with two-version diversity (Technical Committee 7, 1988).

Deficiencies: It is difficult to develop correct but dissimilar specifications for a safety bag and for the main computer and also to avoid common design or implementation decisions. Since there is no fault masking, the safety bag and the main computer must be operational to maintain their availability. Development and implementation of two systems increases overall development and maintenance costs. There are potential problems with information exchange between the two systems and with voting. There is no protection against failures in the safety bag (Technical Committee 7, 1988).

Interrelationship with other capabilities: The safety bag is a form of diversity and is related to assertion programming. It also is similar in some respects to the recovery block scheme (without the recovery).

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.3.5 Sneak Circuit Analysis

A sneak circuit path analyzes defects for an unexpected path or logic flow in a system that may cause a failure or an undesired result. A sneak circuit path may consist of hardware, software, operator actions, or any combination of these items. Sneak circuits are latent conditions, inadvertently designed into the system or coded into the software, that can cause a system to malfunction under certain conditions. The analysis to detect a sneak circuit is based on the recognition of basic topological patterns in hardware or software structure. The analysis uses a checklist of questions regarding the use of, and relationships among, the basic topological components.

Benefits: Sneak circuit analysis can be applied to hardware, software, and the integrated system. It can be applied to programs written in any language, and is directed toward finding system design flaws rather than component failures (Technical Committee 7, 1988).

Deficiencies: This method is labor-intensive. If it is performed late in the development cycle, required changes will be expensive to make (Technical Committee 7, 1988).

Interrelationship with other capabilities: It is complementary to FMEA and Fault-tree Analysis.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.3.6 Retry Fault Recovery

When a system using this technique detects a fault or error, it resets itself to its previous state and re-executes the same code. There are three general categories of methods used to recover to a previous state (Lawrence, 1991): (1) checkpointing, (2) audit trails, and (3) recovery cache. Checkpointing saves enough information to reconstruct the last known correct system state in a disk file. Audit trails save all changes to the system in a transaction log. A checkpoint also is performed. During system recovery, the checkpoint is used to reset the system, and the transaction log is then used to bring the system forward to its last safe state. This technique is used extensively in database management systems. Unlike the checkpointing method, the recovery cache incrementally copies only those portions of the system state that have changed. The system can be restored from this information. Transaction logs also can be used with recovery cache to restore the system to its most current safe state. For real-time components, the software must be retried within its time-critical period. Retry techniques are used frequently in communications systems. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

Benefits: Usually this method is relatively inexpensive to implement. Under normal, correct, operating conditions, minimal software overhead is incurred (Technical Committee 7, 1988).

Deficiencies: Most real-time systems cannot accommodate the loss of data and the time delay incurred in using this technique. Therefore this method is mainly used in handling communication error recovery (Technical Committee 7, 1988) and in database systems.

Interrelationship with other capabilities: This method normally uses software time-out checks or watchdog timers to trigger retry attempts.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.3.7 *n*-Version Programming

In *n*-version programming, multiple development teams use the same software requirements specification. Such programming is an approach to software diversity, and it is used to improve software reliability. Independent teams produce a specified number *n* of software products called versions. The number of software versions is usually three or a greater odd number (Lawrence, 1991). However, for systems that have a safe state, two-version diversity can be used with a bias toward the safe state (Technical Committee 7, 1988). All *n*-versions of the software product are part of the software system. When possible, the software versions use different programming languages and algorithms. By using different programming languages and thus different compilers, it is possible to eliminate compiler errors and unsafe language constructs that may cause common-model failures.

If the various versions produce different outputs, voting is used to select the preferred answer. Different voting strategies can be used, depending on the application requirements. If the system has a safe state, then it is feasible to demand complete agreement of all *n* versions; otherwise, a fail-safe output value is used. For simple trip systems, the vote can be biased in the safe direction. In this case, the safe action would be to trip if either version demanded a trip. This approach typically uses only 2 versions (n = 2). For systems with no safe state, majority voting strategies are employed. For cases where there is no collective agreement, probabilistic approaches can be used to maximize the chance of selecting the correct value.

Benefits: If the software faults produced by one software developer are independent from those of another developer, *n* versions of the software product are unlikely to have the same faults; thus, a developer can reduce the common-mode failures that result from embedded faults in replicated modules (Leveson et al., 1990).

Deficiencies: The cost to implement *n*-version software can be *n* times greater than the cost to develop a single product. Because all teams use the same software specifications, errors in the specifications may cause similar faults in some or all the versions. IEC 65A(Secretariat)122 and experts agree that for this reason, independence among the *n* versions cannot be expected. Software diversity is difficult to implement and can introduce complex recovery problems into the system. Also, the different software products must have additional functionality to synchronize the various versions and to prevent serious errors from causing the computer's operating system to fail. The recovery of systems using *n*-version programming is a research topic at this time.

Interrelationship with other capabilities: The safety bag is related to this technique, as are other methods of design diversity. Components proven safe and reliable by the use of formal methods or their operational usage may reduce the need for diversity in systems. As an example, three different sorting algorithms are not needed for diversity if one sorting algorithm has the desired properties and has been shown to be correct.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.3.8 Recovery Block Programming

Recovery block programming can be considered another approach that uses software diversity to achieve a fault-tolerant system. Several software modules (usually four or more) are implemented independently, each intended to perform the same function. The first or primary software module is executed first. The software then performs self-checking to verify the correctness of the primary module's results. If the self-checking tests are passed, the system proceeds. If not, the other software modules (referred to as first, second, third, etc.) are executed one at a time until one module passes the self-checking tests.

Since it may be necessary to undo the effects of a software component, this technique is applicable only where side effects of the component can be reversed.

Benefits: This technique provides a form of diversity at the program, procedure, or module level.

Deficiencies: The acceptance test is the critical aspect of this technique. The acceptance test can become as complex as the source code algorithm that it is verifying. Reversing side effects may require special hardware (Technical Committee 7, 1988).

Interrelationship with other capabilities: This technique provides software diversity similar to the safety bag and *n*-version programming techniques.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.3.9 Design Metrics

Metrics evaluate structural properties of the software and relate that information to reliability or complexity. Software tools are required to evaluate the measures in order to analyze the source code. The measures can include (1) the complexity of program control, (2) the number of ways a module can be initiated, (3) the number of entries and exits per module, and (4) the program length (established by counting the number of operands and operators).

Benefits: Static source-code metrics are easy to obtain by using readily available commercial software tools. There is some correlation between complexity (measured in terms of path complexity) of a program module and its maintainability and testability.

Deficiencies: Most accepted metrics are applied to sequential, non-concurrent programs. Most real-time systems are inherently concurrent.

Interrelationship with other capabilities: Metrics may be useful in deciding which program modules to test more heavily or to replace.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: Number of Entries/Exits per Module (I).

9.3.10 Petri Nets

Petri nets are used to analyze a system's safety and operational requirements. A Petri net is a graph-theory model that expresses concurrency and asynchronous behavior as information or control flows. Petri nets can be extended to include timing and data-flow features. However, classical Petri nets include only control-flow features. Petri nets are composed of places and transitions. Places are marked or unmarked. Transitions are enabled when all the input places are marked. Once a transition is enabled, it is permitted to "fire," or execute. Once a transition is executed, the inputs are unmarked, and the outputs of the transition are marked.

Benefits: Petri Nets are easy to understand and can provide a thorough analysis of small systems. The model is potentially executable, a feature that can be used to validate the adequacy of the system. Some classes of faults can be modeled together with their impact on safety (e.g., whether a hazardous state can be reached). Petri nets can provide guidance for system modifications to tolerate faults that are discovered (Technical Committee 7, 1988).

Deficiencies: Except for very simple Petri nets, the properties of the whole system are not obvious from a graph's appearance. Analytic methods work well on small networks or on small parts of large networks, but are computationally difficult to manage for large systems. Only a small proportion of the overall set of states can be tested; therefore the direct execution of a program has a limited capacity to demonstrate safety.

Interrelationship with other capabilities: Temporal logic, calculus of communicating systems, and communicating sequential processes can be used as alternatives to Petri nets (Technical Committee 7, 1988).

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: No metrics were found that apply to this technique.

9.4 Implementation Phase

The standards and literature reviewed for this report did not recommend any specific software reliability techniques for the implemenation phase of the life cycle. However, there are several good and practical techniques, such as the use of coding standards, that can improve software quality. Many standards, such as IEC 880 and MoD 00-55, present guidelines or requirements for coding standards. Software unit testing performed immediately after the code module has been implemented can be viewed as an implementation phase technique. In this report, software unit testing is described in Section 9.5.

9.5 Test Phase

9.5.1 Certifying Acquired or Reused Software Reliability

MoD 00-55 requires that any existing library or other software developed outside the software project conform to the requirements of MoD 00-55 and MoD 00-56. IEEE P1228 does not place as stringent restrictions on acquired and reused software, but does require that certification of the previously developed and purchased software be verified to conform to published specifications. MIL-STD 882B requires that acquired and reused software be subjected to the same hazard analysis as newly developed components. Both IEEE P1228 and MIL-STD 882B require that acquired and reused software be tested with newly developed safety-related components. Verification of acquired or reused software can include formal specification, formal design, static source code analysis, and dynamic testing. The technique is further discussed by Everett (1990) and Musa and Everett (1990). Reliability calculations on a system must consider the reliability of all its software. This includes any reused software components or modules, operating systems, communication interface software, and software obtained from outside sources. The software industry is increasingly reusing software developed for one product in new products. Industry experts believe that reused software is more reliable because it has a wide user base, which uncovers more hidden faults. Even so, the reliability of reused software may not be sufficient to satisfy the reliability goals of the new product (Everett, 1990). Confidence in reliability can be increased by performing tests based on the operational profile of the new product (Musa and Everett, 1990)

Benefits: Reusing software components can reduce development and maintenance costs. New products that embody reused software components are in general more trustworthy.

Deficiencies: A software component reused several times in the same or related systems entails a risk of common-mode failure, as all such components have identical faults.

Interrelationship with other capabilities: The operational profile can be used to determine tests for verifying acquired and reused software.

Addressed by: IEEE P1228 and MoD 00-55.

Related Metrics: No metrics were found that apply to this technique.

9.5.2 Reliability Growth Modeling

IEC 1014 is the only standard that directly addresses reliability growth modeling. This technique also is a topic in many publications, such as Everett (1990), Lawrence (1991), and Musa, Ianinno, and Okumoto (1988). Most software products are changed during their operation and maintenance phase to correct faults and to extend or add functionality. After a software component has been modified or developed, it enters a testing phase for a specified time. Failures will occur during this period, and software reliability can be calculated from various measures such as number of failures and execution time to failure. Software reliability is then plotted over time to determine any trends. The software is modified to correct the failures and is tested again until the desired reliability objective is achieved.

Reliability growth modeling is widely used in industry to determine the date of product release for both new and revised software. IEC 1014 recommends that mathematical modeling should start only after a statistically significant number of failures has occurred. Several mathematical models have been developed. Reliability growth models are usually based either on execution time or calendar time. It is generally accepted by software reliability experts that execution-time models are superior to calendar-time models (Musa, Ianinno, and Okumoto, 1988). Jelinski and Moranda developed the first model in 1972, since revised by Shooman and Musa. Other reliability growth models are known as Bayesian, Jelinski and Moranda, Littlewood, Bayesian-Littlewood, Littlewood and Verrall, Keiller and Littlewood, Weibull order statistics, Duane, Goel-Okumoto, Littlewood NHPP, and Schick-Wolverton. Experts

disagree as to the relative usefulness of these models in safety-related systems such as those used in nuclear power generating stations.

Benefits: Reliability growth modeling is compatible with the procedures used by software developers during the test phase. A variation of this modeling technique has been used by software developers for many years. Reliability growth models can be used to form an opinion on a software product's reliability. Therefore they can assist project managers in their decision to proceed to the next development stage (Technical Committee 7, 1988).

Deficiencies: If this is the only technique used to measure software reliability, the costs to achieve the desired reliability late in the software life cycle may be high. Also, software developers may release a product before its performance reaches the desired reliability goal in order to stay within budget and to maintain the schedule. This technique is based on measuring the number of failures caused by software faults. It does not consider impact on safety. The sheer diversity of the various software reliability growth models tends to reduce the user's confidence in any of them (Technical Committee 7, 1988). All of these models use assumptions (faults are independent, test samples are random, and corrections introduce no additional faults) that may not be realistic in a typical software development environment.

Interrelationship with other capabilities: The output from a reliability model graph can be compared with the reliability time line. Reliability tests can be performed with relative frequencies that match the operational profile.

Addressed by: IEC 1014.

Related Metrics: Fault density (I), Cumulative Failure Profile (I), Defect Indices (I), Error Distributions (I), Software Maturity Index (I), Software Purity Level (P), Estimated Number of Faults Remaining (P), Reliability Growth Function (P), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure, Failure Rate (P), Software Release Readiness (I), Test Accuracy (P).

9.5.3 Software Testing

Testing is the process of executing a software product, or portion thereof, to detect its faults. Since testing uncovers a software product's faults, corrections should improve the reliability and thus the availability of the system. However, testing alone seldom adds sufficient reliability to a software product. Reliable software requires good processes throughout all its life cycle phases.

While most standards approach software reliability through quality- enhancing activities in each life cycle phase, one standard, IEC 1014, focuses principally on software testing. IEC 1014 states that reliability measures of both hardware and software can be obtained only through observation, monitoring, and recording of failures. Therefore, from IEC 1014's perspective, the degree to which reliability is improved is determined by the ability of the tests to expose weaknesses. Testing should therefore be as comprehensive as possible to include all peculiar and unforeseen conditions or combinations of conditions that may occur. Testing to this degree may be impractical. However, an operational profile (Section 9.2.1) uses a minimal number of cases to increase the adequacy of testing .

In practice, software module and system testing is the primary method used to determine reliability. Types of testing include unit or module, software integration, functional, regression, system, acceptance, and installation.

These testing types can be categorized by the amount of knowledge each requires about the internal structure of the software product or component being tested. White-box testing, sometimes referred to as glass-box testing, requires a high level of knowledge, whereas black-box testing requires no knowledge. Software integration testing forms the bridge between white- and black-box testing .

In the following section the various testing techniques are grouped under (1) unit or module testing, (2) functional testing, (3) software integration testing, and (4) system testing. The category of testing (white or black box) is noted in parentheses following the title of each subsectrion. The remaining three categories of testing—regression, acceptance, and installation—are discussed under the appropriate life cycle phase. Several standards address testing in general and as a method of improving the quality of the delivered software. These include ASQC Q93, FIPS 101, and IEEE 0829 and 1008.

9.5.3.1 Unit or Module Testing

Unit testing is applied to a separately testable software unit or group of related units. Test cases that exercise specific aspects of a single software unit or module are executed. These test cases are chosen to include a large fraction of input elements. Various levels of testing are performed on the basis of the confidence level needed for the software product. Unit testing, sometimes called white-box testing, generally uses a white-box method. However, individual units or modules can be tested for conformance to their specific functional task. As this testing requires little or no knowledge of the internal code structure, it is a black-box method. Both types of unit testing are discussed in this section.

9.5.3.1.1 Structural Unit Testing (White-Box)

Structural testing is a unit or module test that uses the white-box method. Coding structures (statements, branches, compound conditions, linear code sequence and jump, and data flow), subroutine calling structure, and all possible paths in the module are executed in tests, depending on the level of testing required. There are three types of structural testing: (1) basis, (2) all-paths, and (3) loop testing. Each of these is briefly described in the following paragraphs. IEC 65A(Secretariat)122 highly recommends structural testing for software with very high safety integrity requirements.

Basis path testing is the simplest white-box approach to exercising the internal paths of a software module. A flow graph of the software program control is developed to show all paths through the software module. Graphical representations of the programing constructs *if, while, until*, and *case* are used to develop a pictorial view of the module control. The flow graph consists of nodes and edges. Flow graph nodes are one or more procedural statements, and edges are the control flow from one node to another. Cyclomatic complexity is used to determine the maximum number of test cases needed to ensure that each statement is executed once. The set of linearly independent paths can then be determined and test cases developed to execute each path in the basis set.

All-path testing is the most complex of the unit or module tests. It exercises every path from the initial entry to the final exit in a software module. All-path testing can require an overwhelming amount of test cases. For example, a simple 100-line Pascal program with a single loop executing at most 20 times and having five decision points within four levels may have 100 trillion paths (Pressman, 1987).

Loop testing focuses on the validity of the loop constructs. It can be used in addition to basis testing to uncover initialization errors, index or increment errors, and boundary errors at the loop limits. For simple loops, tests should be developed that skip the loop entirely, make only a single pass through the loop, and then make two passes through the loop. For nested loops, test cases should start at the innermost loop and set all other loops to their minimum values. Simple loop tests are then done on the innermost loop while the other loops are at their minimum values. The innermost loop should then be tested with out-of-range values. Additional test cases are derived by applying this procedure to the next innermost loop and continuing until all loops have been tested.

Benefits: Structural testing is useful in discovering design and implementation flaws. Generation of test cases can be automated. Structural testing generally has the highest error yield of all testing techniques (Humphrey, 1989). Properly executed unit tests can detect as much as 70 percent of the defects in a module (Thayer, Lipow, and Nelson, 1978).

Deficiencies: Structural unit or module testing requires knowledge of a software component's structure. Therefore this testing is generally performed by the developers of that software component. This practice can introduce testing bias. Even if automated tools are used, this technique may require an unacceptable amount of testing time for high-confidence- level software such as that used in nuclear power generating stations.

Interrelationship with other capabilities: This technique should be used to complement functional requirements testing.

Addressed by: IEC 880 and 65A(Secretariat)122, MIL-STD 882B, and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Minimal Unit Test Case Determination (I), Cyclomatic complexity (I), Test Coverage (I).

9.5.3.1.2 Functional Unit Testing (Black-Box)

In functional unit testing, individual units or modules can be tested for conformance to their specific functional task. As this procedure requires little or no knowledge of the internal code structure, it is a black-box method. An example is a single module that transforms data from a text file into a formatted report. Another example is a module that uses multiple input values and a mathematical algorithm to produce a single output term. In either case, no specific knowledge of the internal structure of the module is needed. The module can be developed by a programmer and then tested by an independent tester whose only information is the set of inputs to and expected outputs from the module. However, some knowledge of the code's modular structure is needed in determining which code module performs the functional task being tested.

Benefits: Functional tasks can be tested on the smallest code unit, thus allowing for simple detection of software faults. Functional unit testing can uncover many software faults prior to functional requirements testing, which is applied at a higher level of abstraction.

Deficiencies: Because this testing technique is usually applied to a single code module, the development programmer is most likely to perform the functional unit test. This increases the risk of bias in the testing process, thus decreasing the effectiveness of the test to uncover software faults.

Interrelationship with other capabilities: Some functional requirements are not explicitly stated in sufficient detail for mapping directly to a single code module. Therefore functional requirements testing should be performed to complement functional unit testing.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: Functional or Modular Test Coverage (I), Test Coverage (I).

9.5.3.1.3 Mutation Testing (White-Box)

Mutation testing is a white-box method performed on individual modules of a software product. It is commonly called error seeding. It can be used in other testing areas such as integration testing. Prior to mutation testing, a software developer purposely inserts errors in source-code modules. This is known as seeding. The source code is then tested. All seeded errors should be found. If only some are found, the test cases are not adequate. The number of remaining real errors can be estimated by the equation

$$\frac{\text{number of seeded errors found}}{\text{total number of seeded errors}} = \frac{\text{number of real errors found}}{\text{total number of real errors}}$$

Benefits: Error seeding determines the effectiveness of the testing strategy

Deficiencies: This technique assumes that seeded errors and real errors are similarly distributed in the software product.

Interrelationship with other capabilities: This technique can be compared with other methods used to measure the effectiveness of testing or with methods such as reliability growth models that determine when the testing phase is completed.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: Estimated Number of Faults Remaining (by Seeding) (P), Test Accuracy (P).

9.5.3.2 Functional Testing

Functional testing focuses on software functional requirements. Functional testing attempts to uncover software faults that include (1) incorrect or missing functions, (2) interface errors, (3) data structure errors, (4) external database access errors, (5) performance errors, and (6) initialization and termination errors. Explicitly stated requirements are necessary for developing complete test cases. Test cases can be generated from equivalence classes and cause-consequence diagrams. Test cases should be selected so that all desired aspects of the specifications are exercised. Test cases also should be selected with the intention of finding software program faults. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

The major functional testing technique is functional specification testing. However, other techniques can be used for functional testing. A brief description of each of these techniques follows. All functional testing techniques—functional specification, stress, boundary value, process simulation, and equivalence class—are black-box methods.

9.5.3.2.1 Functional Specification Testing (Black-Box)

Functional specification testing is a black-box method that exercises specified functional requirements of the software. The goal of functional specification testing is to determine whether there are any program faults in meeting particular aspects of the software or system specifications. Test cases are generated from the software and system specifications. These test cases can therefore be created prior to code implementation. Functional specification testing differs from functional unit testing in that no knowledge of a software product's modular structure is needed to perform the tests, which may execute multiple code modules.

Benefits: Since little knowledge of the internal structure of the software is needed, independent personnel can perform these tests. Thus a biased developer does not influence the results. As software is tested in its completed form, any reused or off-the-shelf software also can be tested.

Deficiencies: If this is the only technique used to increase software reliability, the costs to achieve the desired reliability late in the software life cycle may be high. It is realistic to develop test cases that cover only a small portion of possible test conditions. Functional specification testing detects nonconformance with software and system specifications; it does not prove correctness or identify unintended functions (Technical Committee 7, 1988).

Interrelationship with other capabilities: This technique should be used to complement unit testing techniques.

Addressed by: IEC 880 and (Secretariat)122 and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Test Coverage (I).

9.5.3.2.2 Stress Testing (Black-Box)

Stress testing is a functional technique that uses the black-box method. The basic idea is to determine system degradation and failure modes. Stress testing places an exceptionally high workload on the system to verify that **it** will function easily under the expected workload and to determine system failure modes. Test cases are developed to stress the system operations. Changes in the system load can be tested and evaluated for possible failures. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements. Stress testing is primarily a system test function but can be used for software by designing test cases that emphasize software functions. Polling and demand functions are tested by increasing the number of input changes per time unit over the expected normal frequency. For database functions, the number of interactions with the database and its size are increased. All applicable input parameters should be tested at their boundary conditions (highest and lowest possible values) at the same time.

Stress testing also includes simulation of abnormal but realistic operating conditions that place an extraordinary workload upon the system. One such condition might be an event in the process being controlled that generates a large number of rapid-order alarms that require the system to initiate many actions over a very short period of time. Another form of stress testing, sensitivity testing, attempts to find singularities in various computational algorithms. In this instance, the data values that caused the singularities are within the expected range of values.

Benefits: System failures modes and performance degradation modes are determined before real-life occurrence of critical events.

Deficiencies: Exhaustive stress testing is usually not practical. It requires the knowledge of a process engineer who can anticipate and develop realistic abnormal-event scenarios. These scenarios must be simulated, a potentially costly process. In addition, there is no method to ensure that all abnormal event scenarios have been identified and tested.

Interrelationship with other capabilities: The system failures and the software faults that produced these failures can be measured and included in the various fault and failure count and tracking techniques.

Addressed by: DoD 2167A, IEC 65A(Secretariat)122 and 1014, IEEE P1228, and MIL-STD 882B.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P), Test Accuracy (P).

9.5.3.2.3 Boundary-Value Testing (Black-Box)

Boundary-value testing is **a** functional testing technique that uses the black-box method. Boundary-value testing of individual software components or entire software systems is an accepted technique in the software industry. Test cases using minimum, maximum, minimum - 1, and maximum + 1 input range values are developed and executed. Other input values that would generate division by zero and inputs of blanks for character data and null values also should be tested. Test cases that use extreme values for input parameters usually produce output parameters with extreme boundary values. If not, additional test cases are needed to force the output parameters to their extremes. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

Benefits: The method checks that the boundaries of the specification input domain coincide with the boundaries of the final program input domain.

Deficiencies: This technique detects software faults and system failures after a software component has been developed, although the entire software product may not be completed. Modifications to correct faults and improve reliability are more costly in this life cycle phase than in earlier phases. Also, it is probably not possible to exhaustively check all boundary conditions.

Interrelationship with other capabilities: As with stress testing, system failures and software faults that produced these failures can be measured and included in the various fault and failure-count and tracking techniques. This technique is a form of specification-based testing.

Addressed by: IEC 880 and 65A(Secretariat)122 and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Failure Rate (P), Test Accuracy (P).

9.5.3.2.4 Process Simulation Testing (Black-Box)

The goal of process simulation is to test software functions and their interfaces with outside components without affecting the real environment. This is a black-box method. A test system, which also can include hardware, is created to duplicate the real environment. The test system simulates the actual input and output for the real system. Any operator manual entry also is simulated.

Benefits: This technique generates a wide range of operational scenarios for which the system is evaluated, including scenarios that are too dangerous or impossible to reproduce in the actual operating environment.

Deficiencies: As process simulation is very expensive, the correctness of the simulator and the accuracy of the simulations must be considered.

Interrelationship with other capabilities: Stress testing requires some level of process simulation.

Addressed by: IEC 65A(Secretariat)122 and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Run Reliability (P), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Testing Sufficiency (P), Mean Time to Failure (P), Failure Rate (P), Test Accuracy (P).

9.5.3.2.5 Equivalence-Class Testing (Black-Box)

Equivalence-class testing is a functional test that uses the black-box method. Many references do not consider equivalence-class testing a separate technique but rather a method for determining a subset of test cases. It is included here as a separate test technique because of its importance in determining test cases for performing functional testing. Large safety-related systems have an enormous numbers of inputs. Testing each of these inputs is impractical. It can therefore be very beneficial to divide the input domain into classes of data and then to use a sampling of inputs based on these classes of data to create test cases. Guidelines are used to establish the equivalence classes. Guidelines may include, for example, the following: (1) for an input condition specifying a range of values, one valid and two invalid inputs should be defined, (2) for a specific value, one valid and two invalid inputs should be defined, (3) for an input condition specifying a single member of a set, one valid and one invalid input should be defined, and (4) for a Boolean input condition, both valid and invalid Boolean input values should be used in defining test cases.

Benefits: As with functional specification testing, little knowledge of the internal structure of the software is needed; therefore independent personnel can perform these tests.

Deficiencies: As equivalence-class testing only uses a sampling of input values, not all input values are tested. This procedure can leave a software fault undetected.

Interrelationship with other capabilities: Equivalence-class testing is similar to functional requirements testing. This technique should be used to complement unit testing techniques. The operational profile can assist in determining the equivalence-class test cases.

Addressed by: IEC 65A(Secretariat)122.

Related Metrics: Functional or Modular Test Coverage (I), Test Coverage (I).

9.5.3.3 Software Integration Testing

Software integration testing is a systematic technique for constructing a software product and uncovering module interface faults. Software modules that have been unit-tested are combined in various methods and tested as a single functioning unit. Modules can be either be included one at a time, as with bottom-up and top-down approaches, or all modules can be tested at one time, as in the big-bang approach.

9.5.3.3.1 Bottom-up Testing (White-Box and Black-Box)

Bottom-up testing merges and tests software modules from bottom to top. Only modules that do not call another module (called terminal modules) are unit-tested in isolation. The terminal module and the module that called it are merged and tested as a unit. This process is repeated until the top -level module is integrated and tested. Bottom-up testing emphasizes module functionality and performance. Since the upper-level module is not tested, a module driver must be created to supply input parameters to the tested modules.

Benefits: No test stubs are needed, and thus no time is required to implement the stubs. Since the lowest-level software modules—which usually contain the most critical functions—are tested first, their faults are found early.

Deficiencies: Test drivers are required to initiate lower-level code modules. Their implementation increases the testing time. In a large software product, many modules must be tested and integrated before a functioning product is available.

Interrelationship with other capabilities: Bottom-up testing uses the unit testing technique for terminal modules. Bottom-up testing is the reverse of top-down testing.

Addressed by: No standard directly addressed the use of this technique.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.5.3.3.2 Top-down Testing (White-Box and Black-Box)

In top-down testing, a software program is merged and tested from the top code module to the lowest terminal modules. Only the top module is tested in isolation. Modules directly called by this module are then merged one by one into the testing process. This process is repeated until the last terminal module has been included and tested. Since only the top module is unit tested, emphasis is on module interfaces. When a module calls a lower-level module that has not been integrated into the testing process, a stub module is used to simulate the output functions of the missing module. A variation of standard top-down testing is modified top-down testing. This technique requires each module to be unit tested in isolation before it is integrated into the standard top-down process.

Benefits: No test drivers are needed. For control systems this is more advantageous, since most drivers are used for interrupt handling, memory allocation, and input and output handling, procedures that can be complex to implement. Interface errors are discovered early in the testing phase. A basic prototype is available after a few modules have been merged and tested.

Deficiencies: The stub module is rarely as simple as a return to the previous module; therefore implementing the stub may be difficult and increase the testing time. Static, wired-in values are typically used for the module's output parameters, leaving the module being tested correct only for that set of output values. Modules are rarely tested thoroughly as soon as they have been integrated into the testing process (Myers, 1976). Since the upper modules of a software system are tested before their lower level modules, they can be implemented before the entire design of the system has been completed. If this occurs, additional premature design errors can be introduced into a software product.

Interrelationship with other capabilities: Top-down testing is the reverse of bottom-up testing. Unit testing is used for the top module in the standard top-down process and for all modules in the modified top-down process.

Addressed by: No standard directly addressed the use of this technique.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.5.3.3 3 Big-Bang Testing (White-Box and Black-Box)

In big-bang testing, each module is unit-tested in isolation. All modules are then merged simultaneously and tested as a single software unit. This is a simple approach and the most common. Unfortunately, big-bang testing has more disadvantages and fewer advantages than other tests. For small and well-designed software products, the big-bang approach is feasible. In large, complex, safety-related systems, however, this approach can be very dangerous.

Benefits: Module stubs and drivers are not required.

Deficiencies: As modules are not integrated until late in the test phase, interface faults may remain undetected for a long time. This may increase the difficulty of software corrections and the costs of removing faults. Since all modules are merged simultaneously, it can be very difficult to determinine the exact module where the fault resides.

Interrelationship with other capabilities: The big-bang testing technique uses unit testing for every module of the software product.

Addressed by: No standard directly addressed the use of this technique.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.5.3.3.4 Sandwich Testing (White-Box and Black-Box)

Sandwich testing uses bottom-up and top-down testing simultaneously. It merges the code modules from the top and the bottom so that they eventually meet the middle. This approach is reasonable for integrating software in a large software product. A modified sandwich-testing approach performs

isolated unit testing on the higher-level code modules before they are used in the top-down testing scheme.

Benefits: Sandwich testing retains most of the benefits of bottom-up and top-down testing. Some, but not as many, module drivers and stubs are neededas in bottom-up and top-down testing. As in the top-down approach, the module interfaces are tested early in the test phase, and a basic prototype is available. The functional specifications of the software product also are tested early, since they are usually implemented in the lower-level modules.

Deficiencies: Code modules near the boundaries of the sandwich layer may not be thoroughly tested (Myers, 1976).

Interrelationship with other capabilities: The sandwich testing approaches uses the bottom-up and top-down testing processes.

Addressed by: No standard directly addressed the use of this technique.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.5.3.4 System Testing

System testing integrates software with other system components such as hardware and external data sources. The primary purpose of system testing is to fully exercise the computer-based system to find discrepancies between the system and its original objectives. This may include performance, reliability, safety, and functional objectives. Though other types of testing are used to detect faults and thus improve reliability, system testing is the most applicable to measure reliability and increase the assurance of safety.

System testing is commonly subdivided into many categories. The number and content of these categories vary depending upon the individual performing the subdivision. Major categories include stress, probabilistic, and performance testing. Probabilistic and performance system testing techniques are discussed in this section. Stress testing is primarily a system test, but since it can be used in the functional testing process it was included in Section 9.5.3.2.2.

9.5.3.4.1 Probabilistic Testing (Black-Box)

Probabilistic testing is also referred to as statistical testing. To quantify software reliability, the software product or its components are tested with both systematic and random approaches. The systematic perspective takes an overall approach to test control, whereas the random perspective is applied to a specific test case. For the random approach, a predicted distribution of system behavior determines the tests to be executed. Unusual states or rarely used portions of the system may not be included in the predicted distribution. For that reason, especially in safety-related systems, careful attention should be given to including test cases that require the system to take action only in rare but significant circumstances. Test cases should avoid impossible or nonsensical events. Since the number of test cases can be large, automatic testing tools are used to provide test input data and to check the output data. Some tests are performed on large host computers that simulate actual operation. IEC 65A(Secretariat)122 highly recommends this technique for software with very high safety integrity requirements.

Benefits: Since a specification is not required, this method can be used on poorly documented systems. Even though some effort is required to select representative test cases, a large number of cases can be produced with little effort. This form of testing is less prone to human bias error. Testing for failure rate measures up to 10^{-4} failures per hour is feasible.

Deficiencies: If tests are performed using computer hardware that does not have the same processor speed as the hardware on which the software is designed to run, timing issues cannot be included. Some experts believe these tests then are not valid. If the test cases are truly random and do not represent common operating modes of the system, then the reliability estimates generated will be misleading. Since the number of distinct combinations of inputs is very large, a huge amount of test data is produced with no assurance of finding all the systematic errors. Purely random test cases are inappropriate.

Interrelationship with other capabilities: This is a form of black-box testing. It should be used to complement functional testing and in conjunction with structural unit testing.

Addressed by: IEC 880 and 65A(Secretariat)122 and MoD 00-55.

Related Metrics: Software Maturity Index (I), Minimal Unit Test Case Determination (I), Run Reliability (P), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.5.3.5.2 Performance Testing (Black-Box)

Performance is a critical requirement in real-time and embedded systems. If the timing of a software component is greater or smaller than expected by another component, the system's synchronization can fail. This may initiate an event that triggers a fault-recovery function or system failure. Performance tests verify (1) response time under varying loads, (2) percentage of execution time in the various program segments, (3) throughput, (4) primary and secondary memory utilization, and (5) traffic rates on data channels and communication links. Some performance testing can also occur in the unit testing processes. However, until the complete system is integrated and tested, performance cannot accurately be tested. Caution should be used in performance unit testing to avoid spending too much effort on fine-tuning a code module that contributes very little to overall system performance. The most productive performance tests are at the subsystem and system levels (Fairley, 1985).

Benefits: Performance testing detects faults that are usually never uncovered with any other type of testing. Results of the performance tests can be used to optimize the system for the best throughput.

Deficiencies: Performance measurements require performance specifications, which are difficult to generate (Humphrey, 1989).

Interrelationship with other capabilities: Performance tests are often used with stress testing .

Addressed by: IEC 880 and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Run Reliability (P), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (I).

9.6 Installation and Verification Phase

9.6.1 System Certification

Certification is the formal process whereby the developer records and assesses whether the system (hardware and software) has met the customer's specifictions. For safety-related systems, certification includes determining that the safety requirements have been met, the safety-related software has achieved the proper software safety integrity level, and that the work associated with the development of the system conformed to the necessary standards and guidelines. For formal certification, the appropriate signatures of the responsible and accountable personnel are required.

Benefits: Legal responsibility of the customer and the developer regarding the safety of the system may be defined by the formal certification.

Deficiencies: Certification of the system is performed in the last development phase of the software life cycle. Costs to correct faults, increase performance, or improve reliability or safety can be considerably higher than in earlier software life cycle phases.

Interrelationship with other capabilities: Some standards and software development environments allow for formal certification during the various software life cycle phases. With some reviews and audits, signatures of responsible and accountable personnel also are a requirement.

Addressed by: IEC 65A(Secretariat)122, IEEE P1228, and MoD 00-55.

Related Metrics: No metrics were found that apply to this technique.

9.6.2 Acceptance Testing (Black-Box)

Acceptance tests, also referred to as qualification tests, are performed in the system's real-world environment to demonstrate to the customer that a system meets functional, operational, and safety requirements. Acceptance testing is a validation process conducted by the customer. The customer designs and writes the test cases to be performed. However, in some instances the software developer's quality-assurance team also performs acceptance testing, and may add unit and functional test cases. Benchmark tests, which measure the results against a predetermined standard of expected outputs and performance, are also executed.

When a product is intended for multiple customers, acceptance testing is often performed by releasing the system to a limited number of field sites for actual operational testing. This practice is called Beta testing. In some environments it is feasible to operate a newly developed system or version in parallel with the older system that it replaces; this practice is called parallel operation. It has the advantage of ensuring an operational system even if the new system fails. However, running multiple systems may be impossible or prohibitively expensive. Test coverage analyzers, timing analyzers, and coding standards checkers are tools that can assist in performing acceptance testing.

Benefits: The end-user tests the system in its actual operational environment. Often the actual operator of the system performs the tests, and therefore may use a data scenario (operational profile) that was not considered during previous testing.

Deficiencies: If discrepancies between the tested system and the requirements are uncovered, they can be quite costly to correct. The planning, creation, and execution of test cases by the customer requires a significant commitment of resources. Unfortunately, some customers do not commit the appropriate level of resources, and acceptance testing becomes ineffective.

Interrelationship with other capabilities: Acceptance testing is performed as a part of system certification.

Addressed by: DoD 2167A, IEC 65A(Secretariat)122, IEEE P1228, MIL-STD 882B, and MoD 00-55.

Related Metrics: Functional or Modular Test Coverage (I), Software Maturity Index (I), Minimal Unit Test Case Determination (I), Run Reliability (P), Software Purity Level (P), Test Coverage (I), Residual Fault Count (P), Failure Analysis Using Elapsed Time (P), Mean Time to Failure (P), Failure Rate (P).

9.7 Operation and Maintenance Phase

9.7.1 Monitoring Degradation of System

A software system should be periodically monitored to estimate its continuing reliability. IEC 880 references recommends periodic testing of operational software to detect any degradation. However, it does not specifically relate this process to software reliability. Everett (1990) and Musa and Everett (1990) discuss this technique in more detail. Reliability tests are used as the monitoring tool. Corrections and upgrades to software, such as updating the operating system, revising database products, or adding communication software, will affect system reliability. Early in a software system's operational phase the number of failures is relatively high, then decreases. Towards the end of its operational life, failures increase again. Thus, graphs of failures usually form a "bathtub-shaped" curve. In analyzing a failure or reliability trend, software developers can predict at what point in the operational phase it is more cost-effective to replace the software than to continue maintenance. This in-service assessment also can determine the need to increase or decrease the frequency of reliability tests.

Benefits: Reliability information can be used to determine when new software is ready for installation or when it is time to upgrade existing software (Musa and Everett, 1990). The severity and frequency of failures estimated from the reliability tests can be used to set priorities for correcting problems and identifying resource needs.

Deficiencies: This technique measures reliability after a software product has been built. Modifications to improve reliability are more costly during this phase of the software life cycle than in earlier phases. In safety-related software systems, reliance on operational-phase information for determining system reliability is not practical, although information acquired in this phase does provide a measurement of the quality of the operation and maintenance processes.

Interrelationship with other capabilities: Information from this process may indicate a need to perform a root-cause analysis.

Addressed by: IEC 880.

Related Metric: Software Release Readiness.

9.7.2 Root-cause Analysis

No standard addresses the use of a root-cause analysis; however, Everett (1990) and Musa and Everett (1990) discuss this technique. If the reliability of a software system decreases during its operation and maintenance, then those processes themselves may be at fault. Several such factors may contribute to degrading software reliability: (1) the customer's definition of failure may differ from the one used by the software developers to produce and test the product, (2) the actual operational profile is not the same as that used in the design and test phases, (3) the original reliability goals may not reflect the requirements of the customer, or (4) the customer may have changed his performance requirements for the software (Everett, 1990).

Benefits: By determining the root cause of unacceptable software reliability, the customer can improve the development processes and techniques and thus can improve future software maintenance and development.

Deficiencies: This analysis is effective only if corrective action is performed to reduce the number of faults introduced in the remaining life of the system.

Interrelationship with other capabilities: This analysis is initiated after monitoring system degradation.

Addressed by: No standards addressed this technique.

Related Metrics: Defect Density (I), Fault Days Number (I).

9.7.3 Regression Testing (White-Box and Black-Box)

Regression testing is conducted to detect software faults introduced by changes or upgrades made during the operational life cycle phase. It is not uncommon for software fault corrections to introduce new faults in seemingly unrelated functions. The correction of one fault may initiate an untested path or function in previously integrated and installed systems, thus creating a system failure. Configuration management errors are sometimes made when new or updated code modules are integrated into unchanged modules, thus introducing incompatibilities between the new and old modules. Regression testing incorporates selected functional, system, and integration test cases to create a regression test suite. Generally, regression testing is performed after software has been modified and before a new release is installed. However, the suite of tests may be run periodically to detect any regression problems related to hardware and software interfaces. The test suite includes a comprehensive set of tests that are run infrequently, a subset of such tests that can be run frequently, or a combination of the two. The latter combination has proven the most effective (Humphrey, 1989).

Benefits: Regression testing detects rare pathological cases that are often overlooked in functional testing.

Deficiencies: Regression testing does not detect software faults until new modules have been integrated into the system. Detection at this stage increases the cost of correcting a fault over correction during unit, functional, or system testing of the new code module(s).

Interrelationship with other capabilities: The test suite is a combination of functional, system, and integration testing techniques. Regression testing is also used during the system integration testing phase.

Addressed by: IEEE P1228.

Related Metrics: All functional, system, and integration testing metrics apply.

10. Other Issues Addressed by Standards

10.1 Man-machine Interfaces

IEC 880 and MIL-STD 882B are the only standards reviewed that address man-machine interfaces relating to system safety and software reliability. MIL-STD 882B recommends a hazard analysis specifically for the software/human interface functions. Several standards are available for designing and testing of human interfaces in safety-related systems. Some of these are DOE 6480.1A, EPRI NSAC-38 and 39, IEC 0960, 0964, 0965, IEEE 0567, and NUREG-0700, 2254, 2496, 3010 and 3331. A review of these standards is not included as part of this report.

10.2 Subcontract Compliance

Subcontractor software development should meet all standards applicable to the primary contractor or vendor. This issue is addressed by DoD 2167A, IEEE 730 and P1228, and MIL-STD 882B.

10.3 Personnel Qualifications and Training

Training of personnel addresses software developers and operational staff. IEC 65A(Secretariat)122 and 1014, IEEE 730 and P1228, MIL-STD 882B, MoD 00-55, and FAA 018A discuss the skills needed by members of software design teams to meet software quality-assurance standards. IEC 880 and IEEE P1228 discuss training the operations staff for power generating stations. IEEE P1228 requires that on-going training requirements for personnel with safety-related tasks be specified in a safety plan.

10.4 Inspection Personnel

In discussing requirements for development and inspection personnel, MoD 00-55 recommends that the same independent safety auditor oversee the development project from its initial stages through its entire life cycle. To ensure independency, MoD 00-55 further recommends a separate contract be issued for the safety auditor. The safety auditor should be commercially and managerially separate from the system project manager to preclude any possible concerns over conflict of interest. Other standards do not identify a single individual to oversee the development process but state what functions the personnel shall perform.

10.5 Configuration Management

Configuration management of source code is required in all standards. IEEE 730 and P1228, FAA 016A, DoD 2167A, and MoD 00-55 and 00-56 extend configuration control requirements to include documentation generated during development. It is essential that documentation on computer-readable media be protected against loss or corruption by maintaining duplicate copies in separate locations.

10.6 Software, Hardware, and Firmware

The level of discussion on the analysis, design, and development of computer hardware varies greatly among industry and government standards. Five of the standards (DoD 2167A, FAA 013B, 016A, and 018A, and IEC 65A(Secretariat)122) include firmware in their scope. IEC 1014 provides guidance on hardware and software with very little distinction and does not address firmware.

11.Conclusion

Relying solely on measurement for assessing the reliability and safety of ultra-reliable software systems is not sufficient. The use of quality- enhancing processes throughout all phases of the software life cycle, together with reliability measurement procedures, provides better assurance of developing a safe and reliable software system.

Quality-enhancing processes include (1) planning software development and specific safety-enhancing activities; (2) requirements definition, which must include requirements for reliability, safety, and performance; (3) a software design process that includes reliability and safety design approaches such as defense in depth, graceful degradation, redundancy, and functional diversity; (4) testing of software modules in the implementation, test, installation, and operation and maintenance phases; (5) configuration management of analysis results, software development documentation, and source code; (6) corrective action procedures; and (7) verification and validation.

This report identifies numerous techniques and measures, some of which are complimentary. The categorizing of hazard severity and software integrity levels in MIL-STD 882B, MoD 00-55 and 00-56, and IEC 65A(Secretariat)122 can be applied to determine which techniques or combination of techniques should be applied.

During our review of the standards, it became apparent that there were five—(IEC 880 and 65A(Secretariat)122 , MoD 00-55 and 00-56, and MIL-STD 882B)—that best apply to safety-related software systems for nuclear power generating stations. Two of these, IEC 880 and MIL-STD 882B, have been approved by their organizations and are accepted within the computer industry. The other three standards, IEC 65A(Secretariat)122, and MoD 0-55 and 00-56, are still in draft form; new revisions were released in 1991. The fact that these new releases are consistent with their previous versions indicates acceptance of the practices they recommend by the user community. IEEE 352 and 577 are very concise; however neither standard addresses software to the extent of the five standards mentioned above.

The proposed draft standard, IEEE P1228, is near a final vote by the IEEE Software Engineering Standards Committee. In its current form it contains information that is applicable to nuclear power generating stations. This standard should be released in final form by late 1992. It is one of two that, when available in final form, should be reviewed again for recommendations applicable to software systems in nuclear power generating stations. The other standard, IEEE 1012, is currently under revision. This standard's focus is verification and validation, using a hazard categorization level similar to that in MIL-STD 882B.

Appendix A. Techniques and Procedures for Quality Assurance Procedures Cross-Referenced to Standards.

Table A1.

Topic	DoD 2167 A	DoD 2168	FAA 013B	FAA 016A	FAA 018A	IEC 880	IEC 1014	IEC 65A() 122	IEEE 279	IEEE 352	IEEE 577	IEEE 730	IEEE P1228	MIL-STD 882B	MoD 00-55	MoD 00-56
Inclusion of firmware	•	•	•	•	•			•								
SQA plans or procedures	•	•	•	•	•	•		•				•	•		•	
Reviews and audits	•	•	•	•	•	•		•				•		•	•	•
Requirements phase	•	•	•	•	•	•		•	•			•	•	•	•	•
Design phase	•	•	•	•	•	•		•				•	•	•	•	•
Configuration management	•	•	•	•	•	•		•				•	•	•	•	•
Error reporting	•	•	•	•	•	•						•			•	•
Verification and validation	•	•	•	•	•	•		•				•			•	•
Software performance	•					•						•				
Man-machine interfaces						•				•				•		•
Language characteristics						•		•								
Techniques and methodologies					•	•		•		•		•				
Coding standards	•					•		•				•			•	
Documentation standards								•				•				
Subcontractor	•											•	•	•		
Software developer training					•		•	•				•	•		•	
Formal methods						•		•							•	

Appendix B. Techniques and Procedures for Software Safety Cross-Referenced to Standards.

Table B1.

Topic	DoD 2167 A	DoD 2168	FAA 013B	FAA 016A	FAA 018A	IEC 880	IEC 1014	IEC 65A() 122	IEEE 279	IEEE 352	IEEE 577	IEEE 730	IEEE P1228	MIL-STD 882B	MoD 00-55	MoD 00-56
General Safety	•					•		•					•	•	•	
Safety Plans													•	•	•	
Hazard Severity Levels														•		•
Software Safety Integrity Levels								•								
FMEA and CCFA								•		•	•					
Risk Management	•											•	•	•	•	•
Software to Monitor Hardware and Software						•		•								
Preliminary Hazard Analysis													•	•		•
System Hazard Analysis														•		•
Operating and Support Hazard Analysis														•		•
Occupational Health Hazard Analysis														•		•

Table B1. (continued)

Topic	DoD 2167 A	DoD 2168	FAA 013B	FAA 016A	FAA 018A	IEC 880	IEC 1014	IEC 65A() 122	IEEE 279	IEEE 352	IEEE 577	IEEE 730	IEEE P1228	MIL-STD 882B	MoD 00-55	MoD 00-56
Software Requirements Hazard Analysis													•	•		•
Software Top-level Design Hazard Analysis													•	•		
Software Detail Design Hazard Analysis													•	•		
Software Code-Level Hazard Analysis													•	•		

Appendix C. Techniques and Procedures for Software Reliability Cross-Referenced to Standards

Table C1.

Topic	DoD 2167 A	DoD 2168	FAA 013B	FAA 016A	FAA 018A	IEC 880	IEC 1014	IEC 65A() 122	IEEE 279	IEEE 352	IEEE 577	IEEE 730	IEEE P1228	MIL-STD 882B	MoD 00-55	MoD 00-56
Concept Phase																
Set Reliability Goals										•	•					
Identify and Categorize Failure Modes										•	•		•	•	•	•
Fault-tree Modeling								•		•	•				•	
Event-tree Analysis								•								
Cause-consequence Diagram								•								
Reliability Block Diagram								•		•	•					
Probability Modeling								•		•	•					
Requirements Analysis Phase																
Operational Profile								•								
Markov Modeling								•		•	•					
Monte Carlo Modeling								•		•	•					
Certifying Tools and Translators								•							•	

Table C1. (continued)

Topic	DoD 2167 A	DoD 2168	FAA 013B	FAA 016A	FAA 018A	IEC 880	IEC 1014	IEC 65A() 122	IEEE 279	IEEE 352	IEEE 577	IEEE 730	IEEE P1228	MIL-STD 882B	MoD 00-55	MoD 00-56
Design Phase																
Determine Influential Factors on Reliability																
Reliability Time Line Modeling																
Fault Detection and Diagnosis								•								
Safety Bag								•								
Sneak Circuit Analysis								•								
Retry Fault Recovery								•								
n-Version Programming								•								
Recovery Block Programming								•								
Design Metrics								•								
Petri Nets								•								
Test Phase																
Certifying Acquired or Reused Software Reliability													•		•	
Reliability Growth Modeling							•									
Software Testing	•					•	•	•					•	•	•	
Installation and Verfication Phase																
System Certification								•					•		•	
Acceptance Testing	•							•					•	•	•	
Operation and Maintenance Phase																
Monitoring Degradation of System							•									
Root-cause Analysis																
Regression Testing													•			

Appendix D. Glossary of Metrics

cause and effect graphing. A graphical technique that assists in identifying incomplete or ambiguous requirements. It is a formal translation of a natural-language specification into its input conditions and expected outputs. The inputs and the effects of their values are identified and **displayed** on a boolean graph.

combined hardware/software system operational availability. A technique for calculating the availability of a computer system that includes both hardware and software. This measure, which determines system availability, is generally derived from the Markov Chain model.

completeness. A measure representing the degree to which the software specification includes all software requirements. Ten parameters are calculated from various measures such as the number of functions not satisfactorily defined, number of functions, and number of calling routines. Each of these parameters is multiplied by a weight between 0. and 1.0, where the sum of the weights must equal 1.0. These parameters and their weights are summed to produce the completeness measure.

cumulative failure profile. Graph of the cumulative number of failures in the integration and acceptance test phases. This technique is used to predict the reliability through the profile of the number of failures detected. In the testing phase, failures (categorized by their severity) are totaled and then plotted over time. The shape of the curve can be used to determine when testing is complete (the curve flatens out because few or no errors are found over a period of time). If the failures are also totaled by module, the curve can identify modules that need further testing. The graph can also show a trend in the length of time required to detect failures; thus the amount of time required to complete testing can be estimated.

cyclomatic complexity. A quantitative number that reflects the relative complexity of a software component. A graph is used to display sequential groups of program statements as nodes and the program flow between the statements as edges. The measure can be calculated by three methods: (1) number of edges – number of nodes + 1, (2) number of regions in the graph, and (3) number of nodes with multiple exit paths.

data or information flow complexity. A quantitative value derived from paths that flow into and out of a software component and the data structures that are used as input and output to the component. This measure is used to evaluate information flow structure, module information flow structure, and the complexity of the module's interconnections. This measure can be weighted by the length of the software component in number of lines of source statements.

defect density. Number of unique defects detected during an inspection process categorized by the severity and class of the defect. A low value may identify a bad process. However, a low value may also indicate a good process and a good product. This measure can be used after design and code inspections to determine whether the inspection process requires further examination.

defect indices. This measure is a relative index measuring the correctness of software throughout development. For each phase in the life cycle, the number of defects are categorized as serious, medium, or trivial. Weights are applied to each of these categories; for example, the weight for serious defect is 10, medium is 3, and trivial is 1. A phase index is calculated by using the number of defects in each category, their weights, and the sum of the defects found during that life cycle phase. A defect index is calculated from the sum of the phase indices weighted by their phase

number (e.g., design = 2, unit test = 3, and integration test = 4) and the size of the product at that life cycle phase.

design structure. Measure used to determine the simplicity of design of a software product and potential problem areas. The design structure is the sum of six derivatives, each weighted by the user's priority: (1) top-down design, (2) module dependence, (3) module dependent on prior processing, (4) database size, (5) database compartmentalization, and (6) single entry and exit per module. Each derivative is calculated from the module or database elements that satisfy a particular condition.

error distributions. Graphs of the number of errors plotted against descriptive characteristics such as life cycle, fault types, severity of fault, phase where the fault was introduced, and preventive measure. An example is the number of faults plotted against life cycle values of concept, requirements, design, implementation, test, operation, and retirement. The graphs can be analyzed for trends and to identify areas where process improvement may be needed.

estimated number of faults remaining. An estimate of the number of faults in a software product. The technique inserts faults (seeding) into the software's components, and the product is then tested. The number of faults remaining can be estimated from the number of unseeded faults found, the number of seeded faults, and the number of seeded faults found.

failure analysis using elapsed time. A failure-analysis technique in which the estimated number of remaining faults and software reliability are used to determine reliability growth. It is used to predict the total number of faults in the software and software reliability, and to estimate the resources needed. It uses severity-level categorization when observing the execution time between failures. This interval, the failure, and any associated failures are recorded. Well-known statistical tests, such as goodness of fit, enable an estimate of the number of remaining faults, MTTF, and reliability.

failure rate. Measures the number of failures per unit of time. It can be used to indicate the growth of software reliability over testing time.

fault-days number. Represents the number of days that faults spend in the software product from their creation to their removal. For each fault detected and removed during any life cycle phase, the number of days from its creation to its removal is determined. If the creation date of a fault cannot be determined, the fault is assumed to have been created at the middle of the phase where it was introduced. This measure is an indicator of the quality of the software development process.

fault density. Ratio of the number of unique faults found in a software component over a given time period to the size of the component. This measure predicts the remaining faults by comparison with the expected fault density, determines whether sufficient testing has been completed, and establishes standard default densities for comparison and prediction. Software reliability may be qualitatively assessed from information on testing sufficiency. In earlier phases of the life cycle, when KSLOC (thousands source lines of code) is unavailable, other normalizing factors, such as thousands of words of prose, can be used. This measure may categorize faults by their severity, failure types (e.g. input, output, or both), and fault types (e.g. design, coding, and documentation).

functional or modular test coverage. Measure used to calculate an index for software test coverage. It is the ratio of the number of software functions or modules tested to the total number of functions or modules. This measure provides a percentage of the software tested at any time.

graph-theoretic complexity for architecture. A basic complexity measure that uses graphical techniques to produce three complexity values: static complexity, generalized static complexity, and dynamic complexity. A static or dynamic graph is created from software components or modules called nodes and their interconnections, called edges. The specific measure is then calculated from the number of edges, the number of nodes, and resource parameters determined by the user.

independent process reliability. A method for computing the reliability of a system composed of noninterrelated components that have no loops.

man-hours per major defect detected. Measures the average number of hours the total inspection team spends preparing for and conducting inspections per defect found. It provides a quantitative figure that can be used to evaluate the efficiency of the design and inspection processes. The measure reflects the ratio of the total number of hours an inspection team prepares for and conducts inspections, including all inspections to date, to the total number of major defects detected.

mean time to discover the next *k* faults. Estimated time (mean time) that will elapsed before some number k of faults are detected. The mean time to failure between failure i and failure $i + 1$ is summed for the next k faults. By determining the length of time to discover the next k faults, an estimate as to when a software product reaches the desired reliability probability can be calculated.

mean time to failure. Average time between system failures. This measure is the basic parameter required by most software reliability models. It can include weighting or categorization of the failures by complexity, severity, or reinsertion rate. Failures times of the software system are recorded, and the time between failures is calculated. If categorization—such as severity—is being used, then the time of occurrence for failure i and failure $i + 1$ of a particular severity is used to calculate the mean value for that category.

minimal unit test case determination. This technique identifies the smallest number of test cases that are needed to cover all independent paths in a software component. This is useful in generating a minimum set of test cases for unit testing. The cyclomatic complexity is used to determine the number of distinct paths. Its graph identifies the nodes to be executed within the path for each test case.

number of conflicting requirements. A mapping technique that identifies the number of software requirements that may conflict with another. Mappings from the same architecture component to more than one requirement and from multiple architecture components to one requirement are examined for inconsistencies.

number of entries/exits per module. The sum of the number of entry and exit points in a software component. This measure is used as a complexity indicator for the software architecture.

requirements compliance. This technique verifies compliance to the requirements by using system verification diagrams. Such diagrams embody a logical interconnection of decomposition elements and can detect inconsistencies, incompleteness, and misinterpretations. A graph of

software components or elements is created by decomposing a system's functional, performance, and other requirements into graphical elements. These elements are connected according to connectivity and reachability rules specified in IEEE 982.2.

requirements traceability. A technique whereby software requirements are mapped onto the software design architecture. This measure aids in identifying missing or additional requirements. It is the percentage ratio of the number of requirements that are met by the software design architecture to the total number of original requirements. If the percentage is less than 100, some requirements were not included; if it is greater than 100, requirements may have been added. If the percentage is not equal to 100%, the requirements and the design architecture should be examined for consistency.

reliability growth function. A functional technique that estimates the reliability of a system during system testing and when the system design changes continuously during the design testing process. This measure provides an estimate of the time when the desired failure rate or fault density will be achieved. It is calculated from the total number of test cases during a specific stage, the total number of successful test cases in that stage, and the total number of stages.

RELY. A table of processes to be implemented, cross-referenced by the level of reliability required and the life cycle. This technique uses five reliability ratings: very low, low, nominal, high, and very high. Depending upon the reliability rating, the processes required for each development phase can be adjusted to include processes for ensuring that the product achieves the desired reliability goal. An example from this table identifies, for a very high reliability rating, the processes in the requirements and product design life cycle phase that should contain detailed verification (e.g., quality assurance, configuration management, and documentation), independent verification and validation of interfaces, and very detailed test plans and procedures.

residual fault count. The number or remaining faults in a software product. It measures software integrity by using the observed time between failure i and failure $i + 1$ at a particular severity level and the number of failures at that severity level during the ith time interval. Jelinski-Moranda and Poisson-process models are implemented to produce this measure.

run reliability. Measures the reliability of a software product in any life cycle where a set of discrete input patterns and states can be identified.

software documentation and source listing. A quality measure of the software documentation and source listing. The technique is used to collect information to identify software maintenance products and their components that may be inadequate for use in a software maintenance environment. Two questionnaires are used to examine the format and content of the software documentation and source listings. The questionnaires focus on modularity, descriptiveness, consistency, simplicity, expandability, and testability. The two questionnaires, *Software Documentation Questionnaire* and *Software Source Listing Questionnaire*, are described in *Software Maintainability Evaluators Guide,* AFOTEC Pamphlet 800-2, Vol. 3, No 2, March 1987.

software maturity index. An indicator of the maturity of a software product. The number of software modules added, deleted or changed is subtracted from the total number of modules in the software product, giving the number of mature modules remaining. The ratio of the mature modules remaining to the total number of modules is then calculated.

software purity level. This measure provides an estimate of the change in the hazard rate of a software product from the time of initial operation to any specific point in time during the operational phase.

software release readiness. A measure used to determine whether a software product is ready to be released to the customer. It assumes that specific issues can be combined to assess the release readiness of a software product and the risk of using the software system. Such issues include functional test coverage, software maturity, documentation quality factors, machine interface, and software source code quality factors. For each issue an associated risk ranging from 0. to 1.0 (where 0 denotes no risk) and a relative weight (sum of weights equal 1.0) is assigned subjectively. A modification factor is also applied to the associated risk. This modification factor, ranging from 0. to 1.0, is based upon the subjective or quantified influence between specific issues. The associated risks, relative weights, and modification factors are subsequently used to determine the risk .

software science measures. A group of measures (e.g., program vocabulary, observed program length, program complexity, and program level) based upon the number of distinct operands and operators and their occurrences. These measures provide an assessment of software complexity, predict the length of a software program, and estimate the time an average programmer will take to implement a given algorithm.

system performance reliability. A measure of the probability that each performance requirement will be met during the operational phase.

test accuracy. An indicator of the effectiveness of the testing phase in detecting software faults. This measure uses the number of seeded faults and the estimated number of detectable seeded faults to calculate test accuracy. The software system or component to be tested is seeded with faults. The system or component is then tested, and the number of seeded faults detected in each testing interval is recorded. A mathematical model is used to calculate the cumulative seeded faults detected across the total testing period. The closer this value is to 1.0, the better the test accuracy.

test coverage. Percentage of test cases or functional capabilities executed during a set of tests. This measure identifies the completeness of the testing process from the developer's and the customer's perspectives.

testing sufficiency. An indicator of the quality of the test-case coverage. This measure compares actual and predicted faults to assess the sufficiency of software integration testing. Total predicted faults should be calculated using either micro quantitative software reliability models based on path structure or Halstead's software science measures. The number of remaining faults is calculated and compared with the total number of faults detected to date. The results determine the sufficiency of the testing process, and appropriate actions can be taken. Actions may include applying either a test coverage measure to determine whether an adequate number or variety of test cases was executed or a fault density measure to determine whether modules have a high number of errors and should be replaced or redesigned.

Bibliography

AFSCP 800-14 (1987), *Air Force Systems Command Software Quality Indicators*, January 20, 1987.

Butler, R., and G. Finelli (1991), "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability," *Proceedings of the ACM SIGSOFT '91 Conference on Software for Critical Systems*, December 1991.

Cullyer, W. J., S. J. Goodenough, and B. A. Wichmann (1991), "The Choice of Computer Languages for Use in Safety-Critical Systems," *Software Engineering J.* **6**, March 1991.

DoD 2168 (1979), *Defense System Software Quality Program* , August 1979.

DoD 2167A (1988), *Defense System Software Development*, February 1988.

Everett, W. (1990), "Software Reliability Measurement," *IEEE J. on Selected Areas in Communications*, February 1990.

FAA 013B (1989), *Quality Control Program Requirements*, September 1989.

FAA 016A (1987),*Quality Control System Requirements*, September 1987.

FAA 018 (1987), *Computer Software Quality Program Requirements*, September 1987.

Fairley, R. E. (1985), *Software Engineering Concepts* (McGraw-Hill).

Humphrey, W. S. (1989), *Managing the Software Process* (Addison-Wesley).

IEC 65A(Secretariat)122 (1989), *Software for Computers in the Application of Industrial Safety-Related Systems*, August 1, 1991. This supersedes IEC 65A(Secretariat)94 I and II, August 1989.

IEC 0880 (1986), *Software for Computers in the Safety Systems of Nuclear Power Stations*.

IEC 1014 (1981) *Programmes for reliability growth.*

IEEE 279 (1971), *Criteria for Protection Systems for Nuclear Power Generating Stations*,.

IEEE 352 (1987), *Guide for General Principles of Reliability Analysis of Nuclear Power Generating Station Safety Systems*.

IEEE 577 (1976), *Requirements for Reliability Analysis in the Design and Operation of Safety Systems for Nuclear Power Generating Stations* , reaffirmed 1986.

IEEE 982.1 (1988), *Standard Dictionary of Measures to Produce Reliable Software*.

IEEE 982.2 (1988), *Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, Tables 5.1-1 and 4.2-1 through 4.2-9 reproduced from IEEE 982.1-1988 with permission of the IEEE, ©1988 by the Institute of Electrical and Electronics Engineers, Inc.

IEEE P1228 (1992), *Standard for Software Safety Plans,* Draft G, January 1992.

Interim Defence Standard 00-55 (1991), *The Procurement of Safety Critical Software in Defense Equipment-Parts 1 and 2,* April 1991.

Interim Defence Standard 00-56 (1989), *Requirements for the Analysis of Safety Critical Hazards*, May 1989.

Keiller, P., and D. Miller (1991), "On the Use and the Performance of Software Reliability Growth Models," *Reliability Engineering and System Safety* **32**.

Lawrence, J. D. (1991), *Tutorial on Software Reliability and Safety Engineering,* Lawrence Livermore National Laboratory, Livermore, CA, draft internal report .

Leveson, N. G., S.S. Cha, J. C. Knight, and T. J. Shimeall (1990), "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. on Software Engineering*, April 1990.

MIL-STD 882B (1984) *System Safety Program Requirements,* March 1984.

Musa, J. (1989)," Tools for Measuring Software Reliability," *IEEE Spectrum*, February 1989.

Musa, J. and W. Everett (1990), "Software-Reliability Engineering: Technology for the 1990s," *IEEE Software*, November 1990.

Musa, J., A. Iannino, and K. Okumoto (1987), *Engineering and Managing Software with Reliability Measures*, (McGraw-Hill, New York, NY).

Musa, J., A. Iannino, and K. Okumoto (1988), *Software Reliability: Measurement, Prediction, Application* (McGraw-Hill, New York, NY).

Myers, G. J. (1976), *Software Reliability: Principles and Practices*, (Wiley-Interscience).

Parnas, D. L. (1990), "Reviewable Development of Safety Critical Software," *Proceedings of the International Conference on Control & Instrumentation in Nuclear Installations*, May 1990.

Pilsworth, R. (1988), "Software Standards for Defense Procurement," *IEEE Computers and Safety* , November 1988.

Pressman, R. S. (1987), *Software Engineering: A Practitioner's Approach*, (McGraw-Hill, New York, NY).

Shooman, M. L. (1983), *Software Engineering: Design, Reliability, and Management,* (McGraw-Hill, New York, NY).

Technical Committee 7 (1988), *Dependability of Critical Computer Systems*, guidelines produced by the European Workshop on Industrial Computer Systems (Elsevier Applied Science).

Thayer, T. A., M. Lipow, and E. C. Nelson (1978), *Software Reliability, A Study of Large Project Reality, Vol. 2*, (TRW series on software technology).